Mathematical Foundations of Logic and Functional Programming

lecture notes

The aim of the course is to grasp the mathematical definition of the meaning (or, as we say, the semantics) of programs in two paradigms: logic programming (a remarkable example is the Prolog programming language) and functional programming (like Haskell or Scala).

Perhaps surprisingly, the mathematical framework for both of these paradigms is more or less the same. Thus, the first part of the course (the one dealing with Logic Programming) is a bit more involved in math – in the second part (dealing with Functional Programming) we will be able to re-use most of the mathematic material covered in the first part.

Semantics of logic programs

As an introduction, we give some examples of logic programs and the intended semantics of them. A logic program is simply a (not necessarily finite!) set of program clauses. That is, there is no particular ordering of the clauses as in the case, say, imperative programs: a program here is just a set of constraints, a set of logical formulas describing the world in which the programming environment tries to derive facts, from a set of known facts, applying a set of inference rules.

A program clause is a formula of the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n \to q$, where each p_i and q are atomic formulas. For those readers not involved with logic: in a structure, each of these p_i and q evaluate to either 0 (false) or 1 (true); the conjunction $p_1 \wedge p_2 \wedge \ldots \wedge p_n$ evaluates to 1 if and only if all the p_i are 1, that is, $\wedge$ is the „minimum" operator[1], and the implication $F \to G$ evaluates to 1 if and only if the value of F is at most the value of G (that is, if F is true, then G has to be true as well). Note that $\wedge$ has a higher precedence than $\to$: we have to evaluate the body $p_1 \wedge \ldots \wedge p_n$ of the clause first, and then compare this value to the value of the head q of the clause.

So a rule, or a formula of the form $p_1 \wedge \ldots \wedge p_n \to q$ basically states that "if all of the statements $p_1, p_2, \ldots, p_n$ hold, then q holds as well", and a program is simply a set of such rules.

It can happen that $n = 0$, that is, a clause can have an empty body which is written as $\to q$. Note that the usage of the logical connectives and the direction of the implication is not consistent in the literature: there are people writing $q \leftarrow p_1 \wedge \ldots \wedge p_n$, or even $q \leftarrow p_1, \ldots, p_n$; in `Prolog`, these rules are written as `q:-p1,...pn` and when the head is empty, then it's `q.`, ending with a period instead of the `:-` sign but this is only a notational difference.

Now when the body is empty, that's evaluated to 1 (for reasons becoming apparent later on), thus if $\to q$ is true, then q is true as well. That's why program clauses having an empty body are called facts and clauses having a nonempty body are called inference rules, usually.

Consider the following example for a first-order logic program consisting of three clauses.

$$\begin{aligned} &\to \mathsf{even}(0)\\ \mathsf{even}(x) &\to \mathsf{odd}(\mathsf{s}(x))\\ \mathsf{odd}(x) &\to \mathsf{even}(\mathsf{s}(x)) \end{aligned}$$

[1] later on, we will be more precise with that and it'll be called the infimum of the values

Given a program, a structure is always some object which assigns "meanings" to the elementary expressions present in the program; in the case of first-order logics, a structure consists of

- an universe, that is, some set A of objects;
- to each n-ary function symbol (i.e. a function symbol taking n inputs), there is an associated function from A^n to A;
- to each n-ary predicate symbol there is an associated predicate, that is, a mapping from A^n to $\{0, 1\}$, the set of truth values.

So the difference between functions and predicates is their output: each of these take n objects as input, but while functions produce an object, predicates produce a truth value. For example, when the set of objects is the set $\mathbb{N} = \{0, 1, 2, \ldots\}$ of natural numbers (note: in this course, 0 counts as a natural number), then addition and multiplication are (binary) functions (mapping pairs of naturals to naturals), the successor function $n \mapsto n+1$ is also a (unary) function, while equality, less-than and is-even are predicates: $=(n, m)$ holds iff $n == m$ (the equality relation has this meaning in every possible structure), $<(n, m)$ holds iff $n < m$ (now the symbol $<$ is not necessarily defined in an arbitrary set: e.g. if we choose the set of complex numbers as universe, then it's not clear how should we interpret $<$), and $\mathsf{isEven}(n)$ is true if and only if n is an even number. So the first two are binary predicates, while the last one is a unary predicate.

It also makes sense to use nullary functions and predicates: a nullary function is simply a constant (mathematically, an $A^0 \to A$ function, but such functions can be identified with their unique image element), and a nullary predicate is a Boolean value. (This will be important later on.)

Then, in turn, in the example above, even and odd have to be (unary) predicate symbols (since their output is fed into an implication, so it has to be a Boolean value), while s and $\mathsf{0}$ have to be (unary and nullary, respectively) function symbols (since their output is fed to some predicate, thus it has to be an object as well). Also, x is an object-valued variable.

Continuing our example program above, we can interpret our program in the structure where the universe is the set $\mathbb{N}$ of natural numbers; s is interpreted by the successor function $n \mapsto n+1$; $\mathsf{odd}(n)$ holds for the number n if n is an odd number; and $\mathsf{even}(n)$ holds for n if n is an even number. Also, let us interpret the constant $\mathsf{0}$ with the number 0. (Probably this is the structure the programmer had in mind; but the virtual machine interpreting the program has no clue about what the programmer had in mind, it can only see the function and the predicate symbols and make formal, symbolic computations using them.)

Then the three rules state that i) 0 is an even number; ii) if x is an even number, then $x+1$ is an odd number; and iii) if x is an odd number, then $x+1$ is an even number. (Note that the variables are implicitly quantified universally in a logic program.)

These statements hold in our structure, so this structure is a model of the program.

Another structure could be the one in which the universe is the set $\mathbb{Z}$ of the integers, 0 is interpreted by the number 0, s is the negation $x \mapsto -x$, $\mathsf{even}(n)$ holds if $n \geq 0$, and $\mathsf{odd}(n)$ holds if $n \leq 0$. Then the clauses formalize the sentences i) 0 is a nonnegative number, ii) if x is nonnegative, then $-x$ is not positive and iii) if x is not positive, then $-x$ is nonnegative. All of the statements hold again in this structure, so this one is also a model of the program.

Yet another structure is the one in which the universe is the set $\mathbb{N}$ of the natural numbers,

s is the doubling function $n \mapsto 2n$, even holds for even numbers, odd holds for the odd numbers. Then the meaning of the clauses is i) 0 is even (which is true), ii) if x is even, then $2x$ is odd (which is false), and iii) if x is odd, then $2x$ is even (which is true). Since the second clause is not satisfied, this one is not a model of the program.

As we can see, there can be many models of a program. The question of semantics is the following:

Among all the models of a program, which one should we choose?

The "chosen" model is called the semantics of a program.

Now the first transformation of a first-order logic program is called a Herbrand extension, which turns the program into a propositional logic program – albeit the size of the resulting program can be infinite (and it is infinite in most of the cases).

In order to define the Herbrand extension, we have to introduce the ground terms first:

Definition: Ground term.

The set of ground terms is the least set such that

- constant symbols are ground terms,
- if f is an n-ary function symbol and $t_1, \ldots, t_n$ are ground terms, then $f(t_1, \ldots, t_n)$ is a ground term.

Basically, any finite string that can be built up starting from constant symbols, applying function symbols (respecting their arity) is a ground term. For example, if f is a unary function symbol, g is a binary function symbol, and 0 is a constant, then 0, $f(0)$, $f(f(0))$, $g(0,0)$ and $g(f(0), f(g(0,0)))$ are ground terms.

Definition: Herbrand extension.

Given a first-order logic program $\mathcal{P}$, its Herbrand extension is the logic program we get by substituting ground terms in place of its variables in every possible way.

Continuing our running example, the ground terms are 0, $\mathsf{s}(0)$, $\mathsf{s}(\mathsf{s}(0))$, and so on, in general terms of the form $\mathsf{s}^n(0)$ (as 0 is the only constant and s is the only (unary) function symbol).

Then the Herbrand extension of our first-order program contains the clauses

$$\begin{aligned}
&\to \mathsf{even}(0)\\
\mathsf{even}(0) &\to \mathsf{odd}(\mathsf{s}(0))\\
\mathsf{odd}(0) &\to \mathsf{even}(\mathsf{s}(0))\\
\mathsf{even}(\mathsf{s}(0)) &\to \mathsf{odd}(\mathsf{s}(\mathsf{s}(0)))\\
\mathsf{odd}(\mathsf{s}(0)) &\to \mathsf{even}(\mathsf{s}(\mathsf{s}(0)))\\
\mathsf{even}(\mathsf{s}(\mathsf{s}(0))) &\to \mathsf{odd}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))))\\
&\ldots
\end{aligned}$$

and much more, clearly an infinite number of them.

The reason why the Herbrand extension is preferred over the original logic program is that the resulting clauses contain only ground formulas (that is, variable-free formulas) and thus the atomic parts (that begin with a predicate) can be viewed simply as Boolean variables.

That is, in the example above, $\mathsf{even}(0)$, $\mathsf{odd}(0)$, $\mathsf{even}(\mathsf{s}(0))$ and so on are actually Boolean variables. Then, a model of the above program becomes simply an assignment of Boolean variables, no fancy structures with some universe and interpretation for the function and predicate symbols are needed. Instead, there might be an infinite set of Boolean variables and an infinite number of clauses – which is still easier to manage in practice.

Also, the original program $\mathcal{P}$ and its Herbrand extension $\mathcal{P}'$ are tightly related: a model of $\mathcal{P}$ can be transformed into a model of $\mathcal{P}'$ and vice versa, so if we can choose a model for the Herbrand extension, we also choose a model for the original program as well.

Viewing the Herbrand extension of our running example, a possible model is the assignment which assigns 1 to the variables $\mathsf{even}(0)$, $\mathsf{odd}(\mathsf{s}(0))$, $\mathsf{even}(\mathsf{s}(\mathsf{s}(0)))$, $\ldots$, and 0 to the others, that is, $\mathsf{odd}(\mathsf{s}^n(0))$ is true if and only if n is odd and $\mathsf{even}(\mathsf{s}^n(0))$ is true if and only if n is even. (The reader is encouraged to check that this assignment indeed satisfies all the clauses above.)

This (Boolean) assignment corresponds actually to one of the two models of the original program we've already seen: to the one in which we set the universe as the natural numbers and interpret s with the successor function, and even, odd are interpreted by the corresponding parity checker predicates.

Also, if we set the universe to be a singleton $\{0\}$, and interpret $\mathsf{s}(0) = 0$, $\mathsf{even}(0) = \mathsf{odd}(0) = 1$ (that is, both predicates hold true for the single element of the universe), then it's also a model of our first-order program; that model corresponds in the Herbrand extension to the satisfying assignment in which we set all the variables to true.

And again, there are many models of the transformed program.

It suffices to give a semantics for propositional logic programs (containing possibly an infinite number of clauses and variables) – first-order programs will get their semantics from that via the Herbrand extension.

Also, it's worth observing that the assignment which sets every variable to 1 is always a satisfying assigment (since if the head of a clause is true, then the clause is satisfied), but it's probably not the one the programmer had in mind.

So our primary aim is the following:

Given a propositional logic program, that is, a (possibly infinite) set $\mathcal{P}$ of clauses of the form

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \to q,$$

where each p_i and q are Boolean variables from a (possibly infinite) set Z, give a "good" model of $\mathcal{P}$ as "the" semantics of $\mathcal{P}$.

Now we'll see that in more complicated cases (in particular, in the case of generalized logic programs) it's not always clear what makes a model "good"...

One common thing shared by the experts of the field is that

A “good” semantics minimizes the truth values.

But in order to understand this sentence, we’ll need to mathematically define what’s exactly “minimized” here.

In general, we’ll work with partially ordered sets, or posets for short:

Definition: Poset.

A relation $\leq$ over a set P is called a partial order on P if it satisfies all the following conditions:

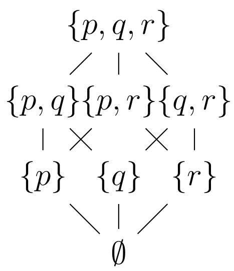
- $x \leq x$ for each $x \in P$ (reflexivity);
- $x \leq y$ and $y \leq z$ imply $x \leq z$ for each $x, y, z \in P$ (transitivity);
- if $x \leq y$ and $y \leq x$ hold for $x, y \in P$, then $x = y$ (antisymmetry).

In this case $(P, \leq)$ is called a partially ordered set, or simply a poset.

If the poset additionally satisfies that for each $x, y \in P$ we either have $x \leq y$ or $y \leq x$ (that’s called dichotomy), then $(P, \leq)$ is a linearly ordered set.

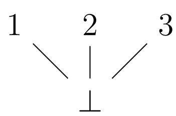
For examples: the set $\mathbb{N}$ of naturals with their standard ordering $\leq$ is a linearly ordered set, and so are the sets $\mathbb{Z}$ of integers, $\mathbb{Q}$ of rationals and $\mathbb{R}$ of reals.

When X is a set, then $P(X)$ is the power set of X, which consists of the subsets of X; then, $(P(X), \subseteq)$ is a poset which is not linearly ordered (apart from the cases when $|X| \leq 1$). For example, we can depict the “Hasse diagram” $P(X)$ with $X = \{p, q, r\}$ as



So, $P(X)$ has eight elements in this case. In a Hasse diagram, $x \leq y$ holds if and only if y can be reached from x via some “elevating” path in the diagram. (Of course there might be problems with this interpretation if the poset is infinite as it’s frequently the case.) Clearly, this $P(X)$ is not linearly ordered, since e.g. $\{p\}$ and $\{q\}$ (and many other pairs) are incomparable elements of the poset, neither of them being a subset of the other one.

Another type of posets is the poset denoted $X_\perp$ where X is some set: in this poset, the underlying set is $X \cup \{\perp\}$ for the new element $\perp$, and the ordering is that $\perp \leq x$ for every member x of the poset, and all the other elements are incomparable. For example, with $X = \{1, 2, 3\}$, the poset $X_\perp$ is



which is also not a linearly ordered poset (again, apart from the cases when $|X| \leq 1$).

We will frequently use two particular posets: the first of them is the poset $\mathbf{2}$ of the Boolean values, with $\{0, 1\}$ as the set and $0 \leq 1$ as the ordering, that is,

$$\begin{array}{c} 1 \\ | \\ 0 \end{array}$$

Clearly, $\mathbf{2}$ is a linearly ordered set (being essentially the same as – isomorphic to – $P(\{1\})$, the power set of a singleton set and also as $\{1\}_\perp$, the pointed poset of a singleton set).

The other poset will be the poset of assignments. Let Z be the set of (Boolean) variables (once and for all – note that it's a set, without any particular ordering!) Then, an assignment is a function $u : Z \to \mathbf{2}$ – a mapping from the set of variables to the set $\{0, 1\}$. In general, when X and Y are sets, then X^Y denotes the set of $Y \to X$ functions, thus $\mathbf{2}^Z$ stands for the set of assignments.

Since $\mathbf{2}$ is a poset, we can turn $\mathbf{2}^Z$ into as poset as well, with the pointwise ordering. In general, when P is a poset and I is a set, then P^I (again: this is the set of functions $I \to P$) is a poset with the ordering $u \leq v$ if and only if $u(i) \leq v(i)$ for all $i \in I$. That is, if u and v are functions, then we say $u \leq v$ if the value $u(i)$ (which is an element of the poset P) is at most the value $v(i)$ for all possible inputs $i \in I$.

Suppose $Z = \{p, q, r\}$. Then the assignments in $\mathbf{2}^Z$ can be represented as vectors of length three: (x, y, z) represents the assignment where the value of p is x, the value of q is y and the value of r is z.

Then, $(0, 0, 1) \leq (1, 0, 1)$ since $\leq$ holds on all three coordinates; but for example, $(0, 1, 0)$ and $(1, 0, 1)$ are incomparable elements (since the first one is greater on the second coordinate, while the second one is greater on the first and the third coordinate). Hence the poset P^I is usually not linearly ordered, even if P is.

In the case $Z = \{p, q, r\}$, the Hasse diagram of the poset $\mathbf{2}^Z$ is

$$\begin{tikzcd}[arrows=-]
 & (1,1,1) \arrow[dl] \arrow[d] \arrow[dr] & \\
(1,1,0) \arrow[d] \arrow[dr] & (1,0,1) \arrow[dl] \arrow[dr] & (0,1,1) \arrow[dl] \arrow[d] \\
(1,0,0) \arrow[dr] & (0,1,0) \arrow[d] & (0,0,1) \arrow[dl] \\
 & (0,0,0) &
\end{tikzcd}$$

which is the same as $P(Z)$! This is true in general: the posets $\mathbf{2}^Z$ and $P(Z)$ are always isomorphic under the mapping $u \mapsto \{p \in Z : u(p) = 1\}$. We will stick to the notation $\mathbf{2}^Z$ since at some point later we'll use logics with three or four possible truth values and it will be more convenient to use notations like $\mathbf{3}^Z$ instead of "hacking" three possible values into the lattice of $P(Z)$.

So, $\mathbf{2}^Z$ is a poset with the pointwise ordering.

We also need the following definitions in order to understand the part of "minimizing" the truth values:

Definition: Minimal and least elements.

If P is a poset and $X \subseteq P$ is a subset of the poset, then $x \in X$ is...

- a minimal element of X if $\forall y \in X\ y \leq x \Rightarrow y = x$; (there is no element of X which is strictly less than x)
- the least element of X if $\forall y \in X\ x \leq y$; ($x$ is less than all the other elements of X)

Dually, $x \in X$ is...

- a maximal element of X if $\forall y \in X\ x \leq y \Rightarrow y = x$; (there is no element of X which is strictly greater than x)
- the largest element of X if $\forall y \in X\ y \leq x$; ($x$ is larger than all the other elements of X)

Note that in a subset X of P there can be many minimal elements. For example, if we have the formula $F = p \vee q \vee r$, then its models are $(1,0,0)$, $(0,1,0)$, $\ldots$, $(1,1,1)$, basically every member of $\mathbf{2}^{\{p,q,r\}}$ is a model of F but $(0,0,0)$. Then, the set of the models of F has three minimal elements, $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$ (since there is no model of F which is strictly less than any of them), but there is no least model of F. Of course it can also happen that a set X has no minimal elements at all (e.g. when our poset is $\mathbf{Z}$ and X is the set of all the negative numbers).

But the least element is always unique (meaning if it exists, then there is only one):

Proposition

If P is a poset, $X \subseteq P$ and x, y are least elements of X, then $x = y$.

Proof

Since x is a least element of X and $y \in X$, we get $x \leq y$. Similarly, since y is a least element of X and $x \in X$, we get $y \leq x$. Applying antisymmetry we get $x = y$.

Also, if there is a least element x, then x is the only minimal element.

So the "minimizing the truth value" part can be formalized as

A "good" semantics of $\mathcal{P}$ is an assignment which is a minimal model (according to the pointwise ordering on $\mathbf{2}^Z$).

So we should construct an assignment u which satisfies all clauses of $\mathcal{P}$ and there is no other model $v \neq u$ of $\mathcal{P}$ with $v \leq u$. This is what we mean by "minimizing truth values". Hence, for our example formula $F = p \vee q \vee r$ (which is not a logic program, only a formula) a good semantics would be one of $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$ but not the other four models.

The function $T_{\mathcal{P}}$

On possible approach to seek for a minimal model is the following:

1. Start from an initial assignment, say $(0, 0, \ldots, 0)$, setting all the variables to 0.
2. Evaluate all the bodies.
3. In the next iteration, set a variable q to 1 if and only if there is a clause which has q as its head and whose body is evaluated to 1 according to our current iteration.
4. Repeat Steps $2 - 3$ till all the clauses are satisfied.

As an example, when we have the following program

$$\begin{array}{cccc} \to p & p \to q & p \to r & p \wedge q \to s \\ t \to s & t \to q & p \wedge t \to s & \end{array}$$

then in the first iteration (let us fix the order (p, q, r, s, t) in assignment) we start from the assignment $(0, 0, 0, 0, 0)$. Then we evaluate all the bodies: only the first rule $\to p$ has a body with value 1 (since empty bodies always evaluate to 1), all the others are false. Since the head of $\to p$ is p, p is set to 1 in the next iteration; all the other variables remain 0. Then our new assignment is $(1, 0, 0, 0, 0)$.

In the next step, the clauses $\to p$, $p \to q$ and $p \to r$ have bodies evaluating to 1, thus their heads, p, q and r are set to 1. Our new assignment $(1, 1, 1, 0, 0)$.

In the next step, $\to p$, $p \to q$, $p \to r$ and $p \wedge q \to s$ have bodies evaluating to 1, thus their heads, p, q, r and s are set to 1. Our new assignment is $(1, 1, 1, 1, 0)$.

In the next step, $t \to s$, $t \to q$ and $p \wedge t \to s$ still have bodies evaluating to 0, our new assignment is still $(1, 1, 1, 1, 0)$.

Actually, $(1, 1, 1, 1, 0)$ is the least model of the program in the example, thus it's the only minimal model, meaning that this is the only possible semantics the program can have.

What we have just done: we iterated some function which took some assignment and produced some other assignment. That is, a function from $\mathbf{2}^Z \to \mathbf{2}^Z$.

Formalizing this function $T_{\mathcal{P}}$ associated to the program $\mathcal{P}$ we get the following:

Definition: The function $T_{\mathcal{P}}$.

$$T_{\mathcal{P}}(u)(q) \ := \bigvee_{p_1 \wedge p_2 \wedge \ldots \wedge p_n \to q \in \mathcal{P}} u(p_1) \wedge u(p_2) \wedge \ldots \wedge u(p_n).$$

That is, if $u \in \mathbf{2}^Z$ is the current assignment, then $T_{\mathcal{P}}(u)$ is the new assignment; the formula says that the value of a variable q in the new assignment should be calculated as follows: first we collect all the clauses in $\mathcal{P}$ having q as head, then we evaluate their bodies (that's the $u(p_1) \wedge \ldots \wedge u(p_n)$ part), and then take the disjunction of the values – if there is at least one evaluated to 1 according to the current assignment, then the new value of q will be 1, otherwise it's set to 0.

Suprema and infima

In the definition of the function $T_{\mathcal{P}}$ we used the symbols $\bigvee$ and $\bigwedge$ – the question is, how should we interpret these operations when there is e.g. an infinite number of values inside the $\bigvee$ (that is, when q appears as the head of infinitely many clauses)? Or the case when we have an empty conjunction (when the body is empty) or an empty disjunction (when there are no clauses having the head q)?

To give a mathematically supported answer to these questions, we recall the following notions:

Definition: Lower and upper bounds, infima and suprema.

When P is a poset and $X \subseteq P$, then an element $y \in P$ is...

- an upper bound of X, denoted $X \leq y$, if $\forall x \in X\ x \leq y$;
- the supremum of X, denoted $y = \bigvee X$, if it is the least upper bound of X;
- a lower bound of X, denoted $y \leq X$, if $\forall x \in X\ y \leq x$;
- the infimum of X, denoted $y = \bigwedge X$, if it is the greatest lower bound of X.

It can happen for a set $X \subseteq P$ that X has absolutely no lower or upper bounds, and also that X does have upper bounds but there is no least upper bound etc.

Consider the pointed poset $\{1,2,3\}_\perp$. There, the set $\{1,2\}$ has the lower bound $\perp$, which is its infimum as well, but there is no upper bound of this set (and thus it has no supremum).

For another example, considering the poset $\mathbb{Q}$ of rationals, equipped with their standard ordering, and setting $X = \{x \in \mathbb{Q} : x \leq \sqrt{2}\}$ as the set of those rationals being smaller than $\sqrt{2}$ (which is known to be an irrational number but it still can be compared to rationals of course), then X has infinitely many upper bounds (any rational number larger than $\sqrt{2}$ is an upper bound) but there is no least of them (among the rationals of course).

Also, if we take the poset $\mathbb{N}$ with the standard ordering, then any nonempty subset $X \subseteq \mathbb{N}$ has a least element, so nonempty subsets have infima. Also, finite sets have suprema (namely, their largest element), but infinite subsets of $\mathbb{N}$ have no upper bounds at all. If we enrich $\mathbb{N}$ with the additional element ∞ and setting $n \leq \infty$ for each $n \in \mathbb{N}$, then the resulting poset $\mathbb{N} \cup \{\infty\}$ is so that every subset has a supremum (finite nonempty subsets' suprema is their largest element; infinite subsets' suprema is ∞ and for $\emptyset$, we will get back to it in a moment).

It is worth to study the case $X = \emptyset$ separately. First, it holds for any $y \in P$ that $\emptyset \leq y$ since the claim $\forall x \in X\ x \leq y$ is vacuously satisfied. Hence every member of the poset P is an upper bound of $\emptyset$, thus the least upper bound has to be the least element of P.

That is, $\bigvee \emptyset$ is always the least element of the poset, which is usually denoted $\perp$; if there is no least element of P, then $\bigvee \emptyset$ does not exist. Similarly, $\bigwedge \emptyset$ has to be the largest element of P.

Thus, the $\vee$ and $\wedge$ operations on $\mathbf{2}$ are actually the supremum and infimum operators: $\vee\{0\} = \vee\emptyset = 0$ and $\vee\{1\} = \vee\{0,1\} = 1$ and similarly, $\wedge\{0\} = \wedge\{0,1\} = 0$ and $\wedge\{1\} = \wedge\emptyset = 1$. This last one explains why should we evaluate empty bodies to 1: an empty conjunction has to have the value 1, and an empty disjunction has to have the value 0. Also, the definition also makes it clear that an "infinitary disjunction" should be handled as follows: first, we collect all the values appearing in the disjunction into a single set, and then we take the supremum of this given set.

We write $X \leq Y$ if $\forall x \in X \forall y \in Y \; x \leq y$, that is, every member of Y is an upper bound of every member of X.

The following fact is easy to check:

Proposition

Assume $X \leq Y$ for the subsets X, Y of P.

i) If $\bigwedge Y$ exists, then $X \leq \bigwedge Y \leq Y$.

ii) If $\bigvee X$ exists, then $X \leq \bigvee X \leq Y$.

iii) If both $\bigwedge Y$ and $\bigvee X$ exist, then $X \leq \bigvee X \leq \bigwedge Y \leq Y$.

Proof

It is clear that $X \leq \bigvee X$ since $\bigvee X$ is an upper bound (namely, the least one) of X by definition and similarly for $\bigwedge Y \leq Y$.

Now let $X \leq Y$ and assume $\bigvee X$ exists. Then for each $y \in Y$, we have $X \leq y$, that is, y is an upper bound of X. Since $\bigvee X$ is the least upper bound of X, we get $\bigvee X \leq y$ for every $y \in Y$, which is the same as writing $\bigvee X \leq Y$.

Similarly, if $\bigwedge Y$ exists, then each $x \in X$ is a lower bound of Y, thus $x \leq \bigwedge Y$, implying $X \leq \bigwedge Y$.

Now if both $\bigvee X$ and $\bigwedge Y$ exist, then we already know $\bigvee X \leq Y$. Applying i) with $\{\bigvee X\}$ playing the role of X there we get $\bigvee X \leq \bigwedge Y \leq Y$.

It is also clear that if one drops several non-maximal elements of a set (in such a way there is at least a retained upper bound of any dropped element), then the set of upper bounds do not change:

Proposition

Suppose X, Y are subsets of the poset P such that for any element $x \in X$ there exists an element $y \in Y$ with $x \leq y$.

Then any upper bound of Y is also an upper bound of X.

Proof

Assume $Y \leq z$. We have to show that $X \leq z$. For this, let $x \in X$ be a member of X. By the condition on X and Y, there is an element $y \in Y$ with $x \leq y$. Since $Y \leq z$, we also have $y \leq z$, and by transitivity we get $x \leq z$.

Thus, in particular, if P has the least element $\perp$, and $X \subseteq P$, then the set of upper bounds of X and $X \cup \{\perp\}$ coincide, and thus if $\bigvee X$ exists, then $\bigvee X = \bigvee(X \cup \{\perp\})$. Indeed: since to each $x \in X$ there is the same x in $X \cup \{\perp\}$, this shows any upper bound of $X \cup \{\perp\}$ is also an upper bound of X; and to each $x \in X \cup \{\perp\}$, either $x \in X$ or $x = \perp$, and in the latter case there is an upper bound of x in X, if X is nonempty. Finally, if $X = \emptyset$, then P is the set of upper bounds of $\emptyset$ and $\{\perp\}$ as well.

Another handy fact on taking suprema is that suprema can be rearranged:

Proposition

Let P be a poset, I and J be index sets, and to each $i \in I$, $j \in J$ let $x_{i,j} \in P$ be an element. Then

$$\bigvee_{i \in I} \bigvee_{j \in J} x_{i,j} = \bigvee_{i \in I, j \in J} x_{i,j},$$

provided all suprema on the left hand side exist.

Proof

Suppose y is an upper bound of the set $\{x_{i,j} : i \in I, j \in J\}$. Then for each $i \in I$, y is an upper bound of $\{x_{i,j} : j \in J\}$ (since this latter set is a subset of the previous one). Since $\bigvee_{j \in J} x_{i,j}$ is the least upper bound of this set, we get $\bigvee_{j \in J} x_{i,j} \leq y$ for each $i \in I$. Hence y is an upper bound of these suprema and hence y is also an upper bound of their supremum, that is, $\bigvee_{i \in I} \bigvee_{j \in J} x_{i,j} \leq y$. Thus, $\bigvee_{i \in I} \bigvee_{j \in J} x_{i,j}$ is a lower bound for each upper bound y of $\{x_{i,j} : i \in I, j \in J\}$, hence in particular

$$\bigvee_{i \in I} \bigvee_{j \in J} x_{i,j} \leq \bigvee_{i \in I, j \in J} x_{i,j}.$$

For the other direction, suppose y is an upper bound of the suprema $\bigvee_{j \in J} x_{i,j}$ for each $i \in I$. Then for each $i \in I$ and $j' \in J$ we have $x_{i,j'} \leq \bigvee_{j \in J} x_{i,j}$ since $x_{i,j'}$ is a member of the set whose supremum is taken on the right side. Thus, $x_{i,j'} \leq y$ as well (since y is an upper bound of $\bigvee_{j \in J} x_{i,j}$) for each $i \in I$, $j' \in J$, implying y is an upper bound of $\{x_{i,j} : i \in I, j \in J\}$ as well. Hence, since $\bigvee_{i \in I} \bigvee_{j \in J} x_{i,j}$ is the least upper bound, we get by choosing $y = \bigvee_{i \in I, j \in J} x_{i,j}$ that

$$\bigvee_{i \in I, j \in J} x_{i,j} \leq \bigvee_{i \in I} \bigvee_{j \in J} x_{i,j}$$

, thus the two values indeed coincide.

Thus in particular, $\bigvee_{i \in I} \bigvee_{j \in J} x_{i,j} = \bigvee_{j \in J} \bigvee_{i \in I} x_{i,j}$ if all the suprema on both sides exist.

Complete lattices

As we have seen, suprema do not necessarily exist. But when they do, that's a "nicely ordered" poset deserving a name:

Definition: Complete lattice.

A poset P is called a complete lattice if every subset of P has a supremum.

For example, **2** is a complete lattice (since we already enumerated all four subsets of **2** and calculated the supremum of each one).

In particular, every complete lattice has a least element, usually denoted $\bot$, since $\bigvee \emptyset$ also has to exist since $\emptyset$ is a subset of the poset, and we already argued that $\bigvee \emptyset$ is always the least element of the poset.

A nice property of complete lattices is that infima also exist:

Proposition

If P is a complete lattice, then each $X \subseteq P$ also has an infimum as well.

Proof

Let $X \subseteq P$ be a subset of the poset. Let $Y = \{y \in P : y \leq X\}$ be the set of the lower bounds of X. Then $Y \leq X$ and since P is a complete lattice, $\bigvee Y$ exists. Thus, $Y \leq \bigvee Y \leq X$. We claim that $\bigvee Y = \bigwedge X$.

Indeed, since $\bigvee Y \leq X$, we have that $\bigvee Y$ is a lower bound of X. But then, $\bigvee Y \in Y$ since Y contains all the lower bounds of X. Thus, since $\bigvee Y$ is an upper bound for Y, we have that $y \leq \bigvee Y$ for each $y \in Y$, along with $\bigvee Y \in Y$ we get that $\bigvee Y$ is the greatest element of Y, that is, the greatest lower bound of X, thus $\bigvee Y = \bigwedge X$.

Of course, $\bigvee P$ is the greatest element of P, so a complete lattice always has a greatest element which is usually denoted $\top$.

Now since **2** is a complete lattice, the right-hand side of the function $T_{\mathcal{P}}$ is well-defined for any program $\mathcal{P}$ since it consists of evaluations, mapping $\mathbf{2}^Z$ to **2**, then taking infima and suprema within **2**, which always exist.

Next, we show that $\mathbf{2}^Z$, the poset of assigments, is also a complete lattice. We do it a bit more general way:

Proposition

If P is a complete lattice, then so is P^I for any index set I.

In particular, suprema are to taken pointwise: if $U \subseteq P^I$ is a set (of functions from I to P), then their supremum is the function $\bigvee U : I \to P$ with $(\bigvee U)(i) = \bigvee_{u \in U} u(i)$ for each $i \in I$.

Proof

Let U be a subset of P^I. Since P is a complete lattice, the suprema $\bigvee_{u \in U} u(i)$ indeed exist for each $i \in I$, since $\{u(i) : u \in U\}$ is a subset of P which always have a supremum in a complete lattice. So let u^* be the function defined as above: $u^*(i) := \bigvee_{u \in U} u(i)$ for all $i \in I$.

We claim that u^* is indeed the least upper bound of U.

First we show that u^* is an upper bound. Let $u \in U$, we have to show that $u \leq u^*$. Since P^I is equipped with the pointwise ordering, this is equivalent to $u(i) \leq u^*(i)$ for each $i \in I$. But this is clear since $u^*(i)$ is the suprema of the set $\{v(i) : v \in U\}$ and $u(i)$ is a member of this set since $u \in U$. Thus u^* is indeed an upper bound.

Next we show that if v is an upper bound of U, then $u^* \leq v$. That is, we have to show

$u^*(i) \leq v(i)$ for each i. Since $U \leq v$, we get that $u \leq v$ for each $u \in U$, yielding $u(i) \leq v(i)$ for each $u \in U$ and $i \in I$.

This means that $v(i)$ is an upper bound of $\{u(i) : u \in U\}$ and since $u^*(i)$ is the least upper bound of this set, we get $u^*(i) \leq v(i)$ for each $i \in I$, hence $u^* \leq v$. Thus u^* is indeed the supremum of U.

Thus, $\mathbf{2}^Z$ is also a complete lattice. (And since $\mathbf{2}^Z$ is isomorphic to $P(Z)$, so is each poset of the form $P(Z)$.)

Models of $\mathcal{P}$ are the pre-fixed points of $T_\mathcal{P}$

The reason why we study the function $T_\mathcal{P}$ is its intimate relation to the set of models of the program $\mathcal{P}$:

Proposition

The assignment $u \in \mathbf{2}^Z$ is a model of $\mathcal{P}$ if and only if $T_\mathcal{P}(u) \leq u$.

Proof

Let $u \in \mathbf{2}^Z$ be an assignment.

Then, u is not a model of $\mathcal{P}$ if and only if there is a clause $p_1 \wedge \ldots \wedge p_n \to q \in \mathcal{P}$ which is false under u.

This is further equivalent to stating that there is a clause $p_1 \wedge \ldots \wedge p_n \to q \in \mathcal{P}$ with $u(p_1) \wedge \ldots \wedge u(p_n) = 1$ and $u(q) = 0$.

This is further equivalent to stating that there is a variable q and a clause $p_1 \wedge \ldots \wedge p_n \to q \in \mathcal{P}$ with $u(p_1) \wedge \ldots \wedge u(p_n) = 1$ and $u(q) = 0$.

This is further equivalent to stating that there is a variable q such that $\bigvee_{p_1 \wedge \ldots \wedge p_n \to q} u(p_1) \wedge \ldots \wedge u(p_n) = 1$ and $u(q) = 0$, since this supremum is 1 if and only if there is a clause whose body evaluates to 1.

By the definition of $T_\mathcal{P}$, this is further equivalent to stating that there exists a variable q such that $T_\mathcal{P}(u)(q) = 1$ and $u(q) = 0$. Since in $\mathbf{2}$ there are only these two possible truth values, this is equivalent to stating that $T_\mathcal{P}(u)(q) \not\leq u(q)$ for some variable q, which means exactly that $T_\mathcal{P}(u) \not\leq u$, as needed.

The property "$f(x) \leq x$" for some function f is again a nice property deserving a name:

Definition: Pre-fixed, post-fixed and fixed points of a function.

When P is a poset and $f : P \to P$ is a function, then $x \in P$ is...

- a pre-fixed point of f if $f(x) \leq x$;
- a post-fixed point of f if $x \leq f(x)$;

- a fixed point of f if $x = f(x)$.

So once again, we can reformulate our aim:

If $\mathcal{P}$ is a logic program, then its semantics should be a minimal pre-fixed point of the function $T_{\mathcal{P}}$.

Indeed: we know that the semantics should be a minimal model of $\mathcal{P}$, and models of $\mathcal{P}$ are precisely the pre-fixed points of $T_{\mathcal{P}}$.

In the following, we will show several smaller facts, which together give us a unique semantics for a logic program:

1. We will show that $T_{\mathcal{P}}$ is a so-called "continuous" function.
2. We will show that whenever $f : P \to P$ is a continuous function, with P being a complete lattice, then f has a least pre-fixed point.

These two statements together with $\mathbf{2}^Z$ being a complete lattice give us the answer, namely: the semantics of $\mathcal{P}$ has to be this least pre-fixed point of $T_{\mathcal{P}}$, since if there is a least pre-fixed point, then it is the only minimal one.

Monotone and continuous functions

There are two important properties of functions, monotonicity and continuity, which give us methods to calculate (pre-)fixed points.

Definition: Monotonicity.

A function $f : P \to Q$ from a poset P into a poset Q is monotone if $x \leq y$ implies $f(x) \leq f(y)$.

The definition of continuity is somewhat more involved. Recall that from calculus, a function $f : \mathbb{R} \to \mathbb{R}$ is called continuous, if the image of some limit is the same as the limit of the images, i.e., $\lim_{n\to\infty} f(x_n) = f(\lim_{n\to\infty} x_n)$, provided the sequence $x_1, x_2, \ldots$ is convergent. Our definition of continuity is very similar to that one: in this field, suprema play the role of limits, so image of the supremum should be the same of the supremum of the images. Instead "convergence" we have "if the supremum exists". Also, the set of real numbers forms a linear order, which is not necessarily true for a poset. Thus, we include an addigional assumption, requiring the "sequence" (rather, a set) to be linearly ordered.

The definition is the following:

Definition: Continuity.

A function $f : P \to Q$ from a poset P into a poset Q is continuous if whenever $X \subseteq P$ is so that

- X is nonempty,
- X is linearly ordered,
- and $\bigvee X$ exists,

then $f(\bigvee X) = \bigvee_{x \in X} f(x)$.

We will shortly see that continuity implies monotonicity. The following observation comes handy in proving this (and a couple of things later):

Proposition

When P is a poset and $x, y \in P$, then $x \leq y$ if and only if $y = \bigvee\{x, y\}$.

Proof

Assume $x \leq y$. Then by $y \leq y$ we have that y is an upper bound of $\{x, y\}$. It is also the least upper bound since if z is also an upper bound in $\{x, y\}$, then in particular $y \leq z$, thus y is a lower bound of any upper bound. Thus, $y = \bigvee\{x, y\}$.

For the reverse direction, if $y = \bigvee\{x, y\}$, then y is an upper bound of $\{x, y\}$, in particular $x \leq y$.

Now we are ready to show that continuity is stronger:

Proposition

If a function $f : P \to Q$ is continuous, then it is also monotone.

Proof

Assume $f : P \to Q$ is continuous and let $x \leq y$ be elements of P. We have to show that $f(x) \leq f(y)$. Now if $x \leq y$, then $\{x, y\}$ is a nonempty (it has either one or two elements, one iff $x = y$), linearly ordered (by $x \leq y$, each pair of elements are comparable of $\{x, y\}$) subset of P, and its supremum exists: $\bigvee\{x, y\} = y$. Then, since f is continous,

$$f(y) \;=\; f(\bigvee\{x, y\}) \;=\; \bigvee\{f(x), f(y)\},$$

so $f(y)$ is an upper bound of the set $\{f(x), f(y)\}$, implying $f(x) \leq f(y)$.

The Tarski Fixed Point Theorem

Having a monotone function $f : P \to P$ is good for various reasons: one of them is that the image of a pre- or post-fixed point remains a pre- or post-fixed point:

Proposition

Assume $f : P \to P$ is a monotone function. If x is a pre- or post-fixed point of f, then so is $f(x)$.

Proof

From $x \leq f(x)$ we get that (applying monotonicity on both sides) $f(x) \leq f(f(x))$, that is, if x is a post-fixed point, then so is $f(x)$.

Similarly, if x is a pre-fixed point, then $f(x) \leq x$, which in turn implies by monotonicity that $f(f(x)) \leq f(x)$, thus $f(x)$ is a pre-fixed point as well.

Given a poset P having a least element $\perp$, it is always a post-fixed point, since $\perp \leq f(\perp)$, whatever $f(\perp)$ is (since $\perp$ is the least element). Applying f on both sides we get $f(\perp) \leq f^2(\perp)$, then again applying f we have $f^2(\perp) \leq f^3(\perp)$ and so on (more formally: from $f^n(\perp) \leq f^{n+1}(\perp)$ we get $f^{n+1}(\perp) \leq f^{n+2}(\perp)$ by one application of f, and since the statement holds for $n = 0$, we get by induction that it holds for all integers $n \geq 0$).

Thus when $f : P \to P$ is monotone for the poset P having the least element $\perp$, we have

$$\perp \leq f(\perp) \leq f^2(\perp) \leq f^3(\perp) \leq \dots$$

(which is usually called an ω-chain).

We claim that the supremum of this chain is the least pre-fixed point of f, moreover, it is even a fixed point:

Proposition: Tarski Fixed Point Theorem.

Suppose P is a poset having the least element $\perp$, $f : P \to P$ is a continuous function and the supremum x^* of the set $\{f^n(\perp) : n \geq 0\}$ exists.

Then x^* is the least pre-fixed point of f, and moreover, it is also a fixed point. (So it is the least fixed point as well, since fixed points are pre-fixed points themselves.)

Proof

First we show that the x^* above is a lower bound for any pre-fixed point. So let x be a pre-fixed point of f. We show that $\{f^n(\perp) : n \geq 0\} \leq x$.

To show this, we have to prove that $f^n(\perp) \leq x$ for each $n \geq 0$, which can be done via induction on n. For the base case $n = 0$ the statement holds since $f^0(\perp) = \perp$, and $\perp \leq x$, whatever x is.

Assume the claim holds for n: $f^n(\perp) \leq x$. Since f is continuous, it is also monotone. Hence we can apply f on both sides and get $f^{n+1}(\perp) \leq f(x)$. But since x is a pre-fixed point, we also have $f(x) \leq x$, thus $f^{n+1}(\perp) \leq x$ also holds, which proves that x is an upper bound of $\{f^n(\perp) : n \geq 0\}$. Since x^* is the least upper bound of this set, we get that $x^* \leq x$, i.e., x^* is a lower bound of any pre-fixed point.

Next we show that x^* is a fixed point of f. We have already seen that $\{f^n(\perp) : n \geq 0\}$ is a linearly ordered (and of course nonempty) set, moreover, its supremum (that is, x^*) exists. Thus, applying continuity of f we get

$$\begin{aligned} f(x^*) &= f(\bigvee\{f^n(\perp) : n \geq 0\}) = \bigvee_{n\geq 0} f(f^n(\perp)) = \bigvee_{n\geq 0} f^{n+1}(\perp) \\ &= \bigvee\{f(\perp), f^2(\perp), \dots\} = \bigvee\{\perp, f(\perp), f^2(\perp), \dots\} = x^*, \end{aligned}$$

so x^* is indeed a fixed point.

(Note that in the last step we used the fact $\bigvee X = \bigvee(X \cup \{\perp\})$ we have seen earlier.)

Hence, if $f : P \to P$ is a continuous function with P being a complete lattice (in which case the supremum above exists) then the least (pre)fixed point of f can be "computed" via a fixed-point iteration: we start from the element $\bot$, and repeatedly apply f on the result of the previous step, finally we take the supremum of all these values (technically, this "computation" may not terminate since we have to produce the infinite sequence $\bot$, $f(\bot)$, $f^2(\bot)$,...first).

We have seen that the least pre-fixed point of the above f is a fixed point as well. This is true in a more general setting:

Proposition

Assume $f : P \to P$ is a monotone function and x is a minimal pre-fixed point of f. Then x is a fixed point of f.

Proof

Since x is a pre-fixed point, we have $f(x) \leq x$. Since f is monotone, appyling f on both sides we get $f(f(x)) \leq f(x)$. Thus, $f(x)$ is also a pre-fixed point of f, moreover, $f(x) \leq x$. Since x is assumed to be a minimal pre-fixed point of f, it has to be the case $f(x) = x$, that is, x is a fixed point.

$T_{\mathcal{P}}$ is continuous

In this part we show that the function $T_{\mathcal{P}}$ defined earlier is a continuous function. We'll break the proof in several parts.

Definition: Projections.

When P is a poset, I is some set, and $i \in I$, then the ith projection from P^I to P is the function $\pi_i : P^I \to P$ defined as

$$\pi_i(u) := u(i).$$

That is, we evaluate the ith coordinate of the function u.

For example, $\pi_2(x, y, z) = y$, $\pi_1(1, 0, 1) = 1$ and $\pi_q(u) = u(q)$, when $q \in Z$ and $u \in \mathbf{2}^Z$. That is, if u is a variable assignment, then a projection is simply the value of a single variable according to the given assignment. Such guys $u(p_i)$ are building block of the definition of $T_{\mathcal{P}}$. And they are continuous:

Proposition

Projections are continuous.

Proof

Let P be a poset, I be a set and $i \in I$. We want to show that $\pi_i : P^I \to P$ is continuous. To this end, let $U \subseteq P^I$ be a nonempty, linearly ordered set of functions whose supremum $u^* = \bigvee U$ exists. We have to show $\pi_i(u^*) = \bigvee_{u \in U} \pi_i(u)$.

But this is clear, since we already know that in P^I, suprema are to be taken pointwise, i.e.,

$$\pi_i(u^*) = u^*(i) = (\bigvee U)(i) = \bigvee_{u \in U} u(i) = \bigvee_{u \in U} \pi_i(u).$$

The next construct used when building up $T_{\mathcal{P}}$ is called a target tupling. Basically, if we have a set of functions $f_i : P \to Q$, $i \in I$ for some index set I, then we can make one single function of them, which outputs the "tuple" or "vector" containing all the results of the f_i functions:

Definition: Target tupling.

If P and Q are posets, I is some set and to each $i \in I$, $f_i : P \to Q$ is a function from P to Q, then their target tupling is the function $\langle f_i \rangle_{i \in I} : P \to Q^I$ defined as

$$\langle f_i \rangle_{i \in I}(x)(i) = f_i(x)$$

for each $x \in P$ and $i \in I$.

For example, when we take the (binary) conjunction and disjunction $\wedge, \vee : \mathbf{2}^2 \to \mathbf{2}$, then their target tupling in this order is the function $\langle \wedge, \vee \rangle$ is a function from $\mathbf{2}^2$ into $\mathbf{2}^2$, defined as

$$\langle \wedge, \vee \rangle(x, y) = (x \wedge y, x \vee y).$$

(Note that this function sorts its input as $(0, 0) \mapsto (0, 0)$, $(1, 0) \mapsto (0, 1)$, $(0, 1) \mapsto (0, 1)$ and $(1, 1) \mapsto (1, 1)$.)

Whenever we have a function $f : P \to Q^n$ for some integer $n \geq 0$ (that is, a function that outputs an n-ary vector), it can always be written as $f = \langle f_1, \ldots, f_n \rangle$, the target tupling of n functions, each f_i being a function from P to Q: f_1 computes the first coordinate of the output, f_2 computes the second, and so on. As in the previous example, the result of sorting two bits is a pair of bits, the first is the smaller value (that's the infimum, computed by $\wedge$), the second is the greater value (computed by $\vee$).

When we evaluate te body of a clause, first we compute the values $u(p_1)$, $u(p_2)$,..., $u(p_n)$, then (say) arrange them to a vector of the form $(u(p_1), \ldots, u(p_n))$ (that's done by target tupling of the projections), and after that we apply the n-ary conjunction function $\wedge_n : \mathbf{2}^n \to \mathbf{2}$.

In the next step we show that the function $u \mapsto (u(p_1), \ldots, u(p_n))$ is continuous:

Proposition

The target tupling of continuous functions is continuous.

Proof

Let $f_i : P \to Q$, $i \in I$ be continuous functions and let $f : P \to Q^I$ stand for their target tupling $\langle f_i \rangle_{i \in I}$.

Let $X \subseteq P$ so that X is nonempty, linearly ordered and has the supremum $x^* = \bigvee X$. We have to show that $f(x^*) = \bigvee_{x \in X} f(x)$. Since f is a function from P to Q^I, these values are functions from I to Q; thus the above equality holds if and only if $f(x^*)(i) = (\bigvee_{x \in X} f(x))(i)$.

From the definition of target tupling we have $f(x^*)(i) = f_i(x^*)$, which further equals to $f_i(\bigvee X) = \bigvee_{x \in X} f_i(x)$ since f_i is continuous. Writing back the definition of target tupling we get this is $\bigvee_{x \in X}(f(x)(i))$ which is the same as $(\bigvee_{x \in X} f(x))(i)$, since in the poset Q^I

suprema are taken pointwise.

In the next step we will apply the n-ary conjunction $\wedge_n$ which is also continuous:

Proposition

The function $\wedge_n : \mathbf{2}^n \to \mathbf{2}$ is continuous for any $n \geq 0$.

Proof

Let $X \subseteq \mathbf{2}^n$ be a nonempty, linearly ordered subset of $\mathbf{2}^n$ having the supremum x^*. We have to show $\wedge_n(x^*) = \bigvee_{x \in X} \wedge_n(x)$.

Since $\mathbf{2}^n$ itself is finite, X is finite as well. Thus, it's a finite, nonempty, linearly ordered set, hence it can be written as $X = \{x_1, \ldots, x_k\}$ with $x_1 \leq x_2 \leq \ldots \leq x_k$. In particular, it has a greatest element x_k, thus $x^* = x_k$.

Since $\wedge_n$ maps into $\mathbf{2}$, we only have to show that $\wedge_n(x_k) = 1$ if and only if $\bigvee_{x \in X} \wedge_n(x) = 1$. But that's clear since $\wedge_n(x_k) = 1$ iff $x_k = (1, 1, \ldots, 1)$. Since x_k is the largest element of X, and the largest element of $\mathbf{2}^n$ as well, this is further equivalent to $(1, 1, \ldots, 1) \in X$, which in turn is equivalent to $\bigvee_{x \in X} \wedge_n(x) = 1$.

Now we know that the function $u \mapsto (u(p_1), \ldots, u(p_n))$ is continuous, and so is $(u(p_1), \ldots, u(p_n)) \mapsto u(p_1) \wedge \ldots \wedge u(p_n)$. The function $u \mapsto u(p_1) \wedge \ldots \wedge u(p_n)$ is the composition of these functions which also preserves continuity:

Proposition

Composition of continuous functions is also continuous.

That is, if $f : P \to Q$ and $g : Q \to R$ are continuous functions, then so is $g \circ f : P \to R$ defined as $(g \circ f)(x) = g(f(x))$.

Proof

Let $X \subseteq P$ be a nonempty, linearly ordered subset of P, having the supremum x^*. We have to show that $(g \circ f)(x^*) = \bigvee_{x \in X} (g \circ f)(x)$.

First, let us observe the subset $f(X) = \{f(x) : x \in X\}$ of Q. Since X is nonempty, $f(X)$ is nonempty as well. Also, if $y_1, y_2 \in f(X)$, then $y_1 = f(x_1)$ for some x_1 and $y_2 = f(x_2)$ for some x_2. Since f is monotone, $x_1 \leq x_2$ implies $y_1 = f(x_1) \leq f(x_2) = y_2$, and similarly, $x_2 \leq x_1$ implies $y_2 \leq y_1$. Thus, since each $x_1, x_2 \in X$ are comparable in P (because X is linearly ordered), each $y_1, y_2 \in f(X)$ are also comparable in Q. Thus, $f(X)$ is also a nonempty, linearly ordered subset of Q.

Now we can proceed as

$$\begin{aligned}
(g \circ f)(x^*) &= g(f(\bigvee X)) \quad \text{, now applying continuity of } f \\
&= g(\bigvee_{x \in X} f(x)) \quad \text{, now since } \{f(x) : x \in X\} \text{ is nonempty,} \\
& \text{linearly ordered, has a supremum, and } g \text{ is continuous,} \\
&= \bigvee_{x \in X} g(f(x)),
\end{aligned}$$

which is exactly we need.

Thus, we have that the functions of the form $u \mapsto u(p_1) \wedge \ldots \wedge u(p_n)$ are continuous. In the definition of $T_{\mathcal{P}}$, the supremum of such functions is taken. This, again, preserves continuity:

Proposition

If I is a set and to each $i \in I$, $f_i : P \to Q$ is a function, then their supremum $\bigvee_{i\in I} f_i : P \to Q$, defined as

$$(\bigvee_{i\in I} f_i)(x) = \bigvee_{i\in I}(f_i(x)),$$

if exists, it is continuous.

Proof

Note that the supremum of functions does not always exist. For example, when $P = \mathbf{2}$, and Q is the pointed poset $\{1,2\}_\perp$, then if $f_1(0) = 1$ and $f_2(0) = 2$, then $(f_1 \vee f_2)(0)$ should be the supremum of $\{f_1(0), f_2(0)\}$, which is $\{1,2\}$ but in $\{1,2\}_\perp$, that supremum does not exist.

But in the current case, when Q is a complete lattice, the supremum always exists.

So assume $X \subseteq P$ is a nonempty, linearly ordered set having the supremum x^*. We have to show $(\bigvee_{i\in I} f_i)(x^*) = \bigvee_{x\in X}(\bigvee_{i\in I} f_i)(x)$. Applying the continuity of each f_i we get

$$(\bigvee_{i\in I} f_i)(x^*) = \bigvee_{i\in I}\bigvee_{x\in X} f_i(x)$$

and

$$\bigvee_{x\in X}(\bigvee_{i\in I} f_i)(x) = \bigvee_{x\in X}\bigvee_{i\in I} f_i(x)$$

by the definition of the supremum function, and these two values coincide as we have seen already.

So far we have shown that all the functions of the form

$$u \mapsto \bigvee_{p_1\wedge\ldots\wedge p_n \to q \in \mathcal{P}} u(p_1) \wedge \ldots \wedge u(p_n)$$

are continuous. These functions map from $\mathbf{2}^Z \to \mathbf{2}$. The function $T_{\mathcal{P}} : \mathbf{2}^Z \to \mathbf{2}^Z$ is actually the target tupling of such functions! To each variable $q \in Z$ we have a function of the above form, and we arrange the results into a "tuple" indexed by Z (that is, we get a function $Z \to \mathbf{2}$, an assignment). Since we have already seen that the target tupling of continuous functions is continuous, we get that $T_{\mathcal{P}}$ is continuous as well.

Summary: Logic Programs

1. A logic program is a set $\mathcal{P}$ of clauses of the form $p_1 \wedge \ldots \wedge p_n \to q$ with each p_i and q being Boolean variables drawn from a set Z. Both Z and the number of clauses in $\mathcal{P}$ can be infinite.

2. A model of the program $\mathcal{P}$ is an assignment $u : Z \to \mathbf{2}$, where $\mathbf{2}$ is the set $\{0, 1\}$ with the ordering $0 \leq 1$ of truth values, which satisfies all the clauses of $\mathcal{P}$. The set of assignments is also denoted $\mathbf{2}^Z$.
3. We associated a function $T_\mathcal{P}$ to a program $\mathcal{P}$, which transforms assignments into assignments: the new value of a variable q is 1 if and only if there exists a clause whose body is 1 according to the old assignment, and whose head is q.
4. We showed that $T_\mathcal{P}$ is a "continuous function".
5. We showed that $\mathbf{2}^Z$ is a "complete lattice".
6. We showed that the models of $\mathcal{P}$ are exactly the "pre-fixed points" of $T_\mathcal{P}$.
7. We argued that we should seek for a "minimal" model of $\mathcal{P}$.
8. We proved the Tarski Fixed Point Theorem which states that there is exactly one minimal pre-fixed point of a continuous function f on a complete lattice; this minimal pre-fixed point is actually a least pre-fixed point, moreover, it is also a fixed point as well and can be constructed as the supremum of the sequence $\perp$, $f(\perp)$, $f^2(\perp)$,..., where $\perp$ denotes the least element of the lattice.
9. Thus, there is only one choice for a "good" semantics of a program $\mathcal{P}$: namely, we start from the all-zero assignment $\perp$, iterate $T_\mathcal{P}$ and take the supremum of the resulting sequence.

Summarizing the math results and introducing a name for this semantics,

To a logic program $\mathcal{P}$, we associate the following function $T_\mathcal{P} : \mathbf{2}^Z \to \mathbf{2}^Z$:

$$T_\mathcal{P}(u)(q) = \bigvee_{p_1 \wedge \ldots \wedge p_n \to q \in \mathcal{P}} u(p_1) \wedge \ldots \wedge u(p_n).$$

Then, the canonical semantics of $\mathcal{P}$ is

$$\bigvee_{n \geq 0} T_\mathcal{P}^n(\perp)$$

with $\perp$ being the all-zero assignment.

The canonical semantics is the least model of $\mathcal{P}$. Moreover, it is additionally a fixed point of $T_\mathcal{P}$. By the way, fixed points of $T_\mathcal{P}$ are called supported models of $\mathcal{P}$.

Generalized Logic Programs

In this part we extend the framework developed in the previous part to generalized logic programs. Such a program is a (possibly infinite) set of clauses of the form

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \wedge \neg q_1 \wedge \neg q_2 \wedge \ldots \wedge \neg q_k \to r,$$

where each p_i, q_j and r are variables, again drawn from a (possibly infinite) set Z. That is, negated variables can appear in the body of a clause but the head is always a (positive) variable.

We want to retain most parts of the previous framework. So, we can define the function $T_\mathcal{P} : \mathbf{2}^Z \to \mathbf{2}^Z$ similarly to the previous case:

$$T_\mathcal{P}(u)(r) = \bigvee_{p_1 \wedge \ldots \wedge p_n \wedge \neg q_1 \wedge \ldots \wedge \neg q_k \to r \in \mathcal{P}} u(p_1) \wedge \ldots \wedge u(p_n) \wedge \neg u(q_1) \wedge \ldots \wedge \neg u(q_k).$$

Again, models of $\mathcal{P}$ are precisely the pre-fixed points of $T_\mathcal{P}$.

However, there are serious problems with continuity and even with monotonicity.

Consider the example program

$$\neg p \wedge \neg q \to r$$
$$\neg q \wedge \neg r \to p$$
$$\neg p \wedge \neg r \to q$$

Then, if we start from the assignment $(0, 0, 0)$ (the ordering of the variables is p, q, r as in the previous examples), then every body gets evaluated to 1 (as $\neg 0 \wedge \neg 0$ is 1), thus the new value of all the variables is set to 1, that is, $T_\mathcal{P}(0, 0, 0) = (1, 1, 1)$ which is a model of $\mathcal{P}$, so far so good.

But then, iterating $T_\mathcal{P}$ once more, all the bodies are evaluated to 0 this time, thus each variable is set to 0 again. That is, $T_\mathcal{P}(1, 1, 1) = (0, 0, 0)$ which is not that good: this $T_\mathcal{P}$ is not a monotone function!

Another problem we face is that this program, viewed as the conjunction of its clauses, is equivalent to the formula $p \vee q \vee r$ – thus it has three minimal models as we have seen in one of our starting examples, there is no least model. However, in this case these three minimal models, $(0, 0, 1)$, $(0, 1, 0)$ and $(1, 0, 0)$ are all fixed points of $T_\mathcal{P}$, so they are supported models of $\mathcal{P}$.

But, considering the even smaller program $\neg p \to p$, in that case $T_\mathcal{P}(0) = 1$ and $T_\mathcal{P}(1) = 0$, thus the only model $p = 1$ is not a fixed point, it's not a supported model.

So the main problems are: $T_\mathcal{P}$ is not always monotone; it does not always have a least pre-fixed point; it does not always have a fixed point at all.

The 4-valued logic

There are more options to resolve these issues. We choose the following path:

Instead of the logical values 0 and 1, we assign intervals of truth values to the variables.

In general, when P is a poset, P^2 can be seen as the set of intervals of P, with (x, y) representing the set $\{z : x \leq z \leq y\}$ of elements of P between the two endpoints[2].

That is,

Definition: Values of the 4-valued logic.

Elements of $\mathbf{4} = \mathbf{2} \times \mathbf{2}$ are denoted as follows:

- The element $(0, 0)$ represents the set $\{0\}$ "can be only false", and is denoted f.
- The element $(1, 1)$ represents the set $\{1\}$ "can be only true", and is denoted t.
- The element $(0, 1)$ represents the set $\{0, 1\}$ "unknown, can be both", and is denoted $\perp$.
- The element $(1, 0)$ represents the empty set, "inconsistent, cannot be assigned", and is denoted $\top$.

The first three elements are called the consistent elements of $\mathbf{4}$.

In general, an element (x, y) of P^2 for an arbitrary poset P is called consistent if $x \leq y$.

We define two partial orders: the truth order $\leq_t$ and the precision order $\leq_p$ on P^2:

Definition: Truth order and precision order.

For a poset P we define the following partial orders $\leq_t$ and $\leq_p$ on P^2:

$$\begin{aligned}(x, y) \leq_t (x', y') &\iff x \leq x' \text{ and } y \leq y' \\ (x, y) \leq_p (x', y') &\iff x \leq x' \text{ and } y' \leq y.\end{aligned}$$

Basically, if $x \leq y$ denotes in P that y is "more true" than x, then $(x, y) \leq_t (x', y')$ denotes (more or less) that the interval (x', y') is more true than the interval (x, y). For the order $\leq_p$, $(x, y) \leq_p (x', y')$ holds if the interval (x', y') is contained inside the interval (x, y), thus, in a sense, (x', y') is a "more precise" interval than (x, y).

We will want to take suprema and infima of subsets of P^2 with respect to both orderings, hence we have to use different notations for these operations in order to be distinguishable.

Definition: Infima and suprema in P^2.

In P^2,

- $\bigvee$ denotes the supremum operation with respect to $\leq_t$;
- $\bigwedge$ denotes the infimum operation with respect to $\leq_t$;
- $\bigoplus$ denotes the supremum operation with respect to $\leq_p$;

[2]Note that x and y are members of (x, y) in this formalism. In calculus, one writes $[x, y]$ for intervals like these

- $\bigotimes$ denotes the infimum operation with respect to $\leq_p$.

The following is clear[3].

Proposition

When P is a poset and $X = \{(x_i, y_i) : i \in I\}$ is a subset of P^2, then

- $\bigvee X = \big(\vee_i x_i, \vee_i y_i\big)$,
- $\bigwedge X = \big(\wedge_i x_i, \wedge_i y_i\big)$,
- $\bigoplus X = \big(\vee_i x_i, \wedge_i y_i\big)$ and
- $\bigotimes X = \big(\wedge_i x_i, \vee_i y_i\big)$.

Proof

The poset $(P^2, \leq_t)$ is simply P^2 with the pointwise ordering, proving the first two items. For the third item, (x^*, y^*) is an upper bound of X if $(x_i, y_i) \leq_p (x^*, y^*)$ for each $i \in I$, that is, $x_i \leq x^*$ and $y^* \leq y_i$ for each $i \in I$. That is, if and only if x^* is an upper bound of the x_i and y^* is a lower bound of the y_i. Hence $\big(\vee_i x_i, \wedge_i y_i\big)$ is an upper bound for X. Also, if (x, y) is an upper bound of X, then $\{x_i : i \in I\} \leq x$, implying $\vee_i x_i \leq x$ and similarly, $y \leq \wedge_i y_i$ also holds, thus $\big(\vee_i x_i, \wedge_i y_i\big)$ is indeed the supremum with respect to $\leq_p$. The fourth item can be proven analogously.

Thus, if P is a complete lattice, then so are $(P^2, \leq_t)$ and $(P^2, \leq_p)$ (since in a complete lattice all infima also exist, thus the right-hand sides of the previous Proposition are always defined). That's why structures of the form $(P^2, \leq_t, \leq_p)$ are called bi-lattices.

Let us visualize the two orderings in **4** (using the aliases we introduced earlier: $\top$ for $(1, 0)$, f for $(0, 0)$ etc):

$\leq_t$ $\leq_p$

The ordering $\leq_p$ makes clear why we use $\top$ for $(1, 0)$ and $\bot$ for $(0, 1)$: these are the greatest/least elements of $\leq_p$, respectively. Also, we use $\wedge$ and $\vee$ for infimum and supremum with respect to $\leq_t$ since this makes $\mathsf{t} \wedge \mathsf{f} = \mathsf{f}$ and so on, so the "usual" semantics of $\vee$ and $\wedge$ are retained for these "point-like" intervals.

We already have the operations $\vee$ and $\wedge$ within **4**, now we'll define negation on intervals. But how should we do that? First, we should have $\neg\mathsf{f} = \mathsf{t}$ and $\neg\mathsf{t} = \mathsf{f}$ since we want to

[3] It would be even more clear if we introduced the product posets $\prod P_i$ and not only the special case P^I. We will see how it goes this way.

extend the negation from **2** to **4**. It also makes sense to define $\neg\bot = \bot$, since if we do not know anything about a variable's value (i.e. it can be either 0 or 1), then we do not know anything about its negation. Similarly, it also makes sense to define $\neg\top = \top$, since if a value is contradictionary, then so is its negation. It can be checked that the following definition accomplishes this:

Definition: Negation on 4.

$$\neg(x, y) = (\neg y, \neg x).$$

Then, we can define the following function $\Phi_{\mathcal{P}} : \mathbf{4}^Z \to \mathbf{4}^Z$ as follows:

Definition: The function $\Phi_{\mathcal{P}}$.

$$\Phi_{\mathcal{P}}(u)(r) = \bigvee_{p_1\wedge\ldots\wedge p_n\wedge\neg q_1\wedge\ldots\wedge\neg q_k\to r\in\mathcal{P}} u(p_1)\wedge\ldots\wedge u(p_n)\wedge\neg u(q_1)\wedge\ldots\wedge\neg u(q_k).$$

That is, syntactically the above function coincides with $T_{\mathcal{P}}$ defined earlier. The difference is the domain: while $T_{\mathcal{P}}$ is a function over classical (binary) truth values, $\Phi_{\mathcal{P}}$ works with assignments that assign intervals to each variable.

We have not defined the "truth table" for implication – it makes sense to define the value of $x \to y$ as $(\neg x) \vee y$, the latter two operations being already defined for intervals. Or, it also makes sense to set $x \to y$ to t if $x \leq_t y$ and f otherwise – both variants extend the classical case. If we choose to do the latter, then again, pre-fixed points of $\Phi_{\mathcal{P}}$, with respect to the truth ordering $\leq_t$, are exactly the models of $\mathcal{P}$.

The current plan

We outline the steps we will make in order to have a semantics for generalized logic programs, with assignments coming from $\mathbf{4}^Z$:

1. We will transform $\Phi_{\mathcal{P}}$ to an equivalent form $\Psi_{\mathcal{P}}$, which will compute the very same function but which is easier to handle mathematically.
2. In particular, we will show that $\Psi_{\mathcal{P}}$ is a monotone function with respect to $\leq_p$. It will not be continuous, though.
3. We will prove the Kleene Fixed Point Theorem, stating that any monotone function $P \to P$ has a least (pre)fixed point when P is a complete lattice. Moreover, this least fixed point can be defined via some kind of fixed point iteration.

Then, we will call this least fixed point of $\Psi_{\mathcal{P}}$ the Kripke-Kleene semantics of the program $\mathcal{P}$. After that,

1. We will prove that the Kripke-Kleene semantics never has inconsistent values, so it's essentially a **3**-valued model.
2. We will outline some problems with the Kripke-Kleene semantics: most notably, it minimizes only with respect to $\leq_p$ but not with respect to $\leq_t$, which contradicts to the "a good semantics minimizes the truth values" rule.

3. We will introduce so-called stabilizer functions. Using these, we get from $\Psi_{\mathcal{P}}$ an "even better" function, also having a least fixed point with respect to $\leq_p$.

4. We will show that this least fixed point of the stabilizer function is also a fixed point of $\Psi_{\mathcal{P}}$ which is also $\leq_t$-minimal.

We will call this least fixed point the well-founded semantics of $\mathcal{P}$, closing the part on Logic Programming.

The function $\Psi_{\mathcal{P}}$

In this section we convert our function $\Phi_{\mathcal{P}}$, which is a $\mathbf{4}^Z \to \mathbf{4}^Z$ to another function $\Psi_{\mathcal{P}}$, a $\mathbf{2}^Z \times \mathbf{2}^Z \to \mathbf{2}^Z \times \mathbf{2}^Z$ function. The intuition is the following: $\Phi_{\mathcal{P}}$ gets as input two separate assignments, u and v, both coming from $\mathbf{2}^Z$. These two assignment together determine an interval-valued assignment in $\mathbf{4}^Z$ in the following way: for a variable $q \in Z$, $u(q)$ gives the "left end-point" of the interval, and $v(q)$ gives the "right end-point" of the interval.

Also, the output value of $\Psi_{\mathcal{P}}(u,v)$ will be a pair (u',v') with u' and v' being assignments, coming from $\mathbf{2}^Z$: u' will be the assignment computing the new left end-points, and v' will be the assignment computing the new right end-points.

Clearly, the function $\Psi_{\mathcal{P}}$ is basically the same as $\Phi_{\mathcal{P}}$, only the domain is transformed into an isomorphic one, from $(\mathbf{2} \times \mathbf{2})^Z$ to $\mathbf{2}^Z \times \mathbf{2}^Z$. But again, the difference is only that while in $\Phi_{\mathcal{P}}$, the variables directly get an interval as value, in $\Psi_{\mathcal{P}}$ these intervals are decomposed to the two end-points.

Since the $\Psi_{\mathcal{P}}$ we want to construct is a function from $\mathbf{2}^Z \times \mathbf{2}^Z$ to $\mathbf{2}^Z \times \mathbf{2}^Z$, in particular, the output of the function is a pair of assignments, $\Psi_{\mathcal{P}}$ is the target tupling of two functions: let us call the function computing the new left end-points $f_{\mathcal{P}}$, and the function computing the new right end-points $g_{\mathcal{P}}$. Then, $f_{\mathcal{P}}$ and $g_{\mathcal{P}}$ are $\mathbf{2}^Z \times \mathbf{2}^Z \to \mathbf{2}^Z$ functions.

Observing the function

$$\Phi_{\mathcal{P}}(u)(r) = \bigvee_{p_1 \wedge \ldots \wedge p_n \wedge \neg q_1 \wedge \ldots \wedge \neg q_k \to r \in \mathcal{P}} u(p_1) \wedge \ldots \wedge u(p_n) \wedge \neg u(q_1) \wedge \ldots \wedge \neg u(q_k),$$

how is the new left end-point of the interval assigned to r calculated? It's the left end-point of the interval

$$\bigvee_{p_1 \wedge \ldots \wedge p_n \wedge \neg q_1 \wedge \ldots \wedge \neg q_k \to r \in \mathcal{P}} u(p_1) \wedge \ldots \wedge u(p_n) \wedge \neg u(q_1) \wedge \ldots \wedge \neg u(q_k).$$

Since in P^2 we have seen that $\bigvee_{i \in I}(x_i, y_i)$ is $\bigvee_{i \in I} x_i$, we have to compute the left end-points of the intervals

$$u(p_1) \wedge \ldots \wedge u(p_n) \wedge \neg u(q_1) \wedge \ldots \wedge \neg u(q_k)$$

and take their supremum. Also, the left end-point of $\wedge$ of intervals is the $\wedge$ of the left end-points of the same intervals, so we have to take the set of left end-points of the intervals

$$u(p_1), \ldots, u(p_n), \neg u(q_1), \ldots, \neg u(q_k)$$

and take their infimum. Now $\Psi_{\mathcal{P}}$ takes as input two functions (assignments from $\mathbf{2}^Z$): u_1 and u_2 such that $u(p) = (u_1(p), u_2(p))$ for each $p \in Z$. Thus, the left end-point of $u(p_i)$ is $u_1(p_i)$ for each $1 = 1, \ldots, n$. What's the situation with the intervals of the form $\neg u(q_j)$? Well, since

$u(q_j) = (u_1(q_j), u_2(q_j))$, we have that $\neg u(q_j) = (\neg u_2(q_j), \neg u_1(q_j))$ (plugging in the definition of negation in $\mathbf{4}$), hence the left end-point of $\neg u(q_j)$ is $\neg u_2(q_j)$. Thus,

$$f_{\mathcal{P}}(u_1, u_2)(r) = \bigvee_{p_1\wedge\ldots\wedge p_n\wedge\neg q_1\wedge\ldots\wedge\neg q_k\to r\in\mathcal{P}} u_1(p_1) \wedge \ldots \wedge u_1(p_n) \wedge \neg u_2(q_1) \wedge \ldots \wedge \neg u_2(q_k).$$

Using an analogous derivation for the right end-points we arrive to the following definition:

Definition: The function $\Psi_{\mathcal{P}}$

Given a generalized logic program $\mathcal{P}$, the function $\Psi_{\mathcal{P}} : \mathbf{2}^Z \times \mathbf{2}^Z \to \mathbf{2}^Z \times \mathbf{2}^Z$ is defined as $\Psi_{\mathcal{P}} = \langle f_{\mathcal{P}}, g_{\mathcal{P}} \rangle$ with

$$f_{\mathcal{P}}(u_1, u_2)(r) = \bigvee_{p_1\wedge\ldots\wedge p_n\wedge\neg q_1\wedge\ldots\wedge\neg q_k\to r\in\mathcal{P}} u_1(p_1) \wedge \ldots \wedge u_1(p_n) \wedge \neg u_2(q_1) \wedge \ldots \wedge \neg u_2(q_k)$$

and

$$g_{\mathcal{P}}(u_1, u_2)(r) = \bigvee_{p_1\wedge\ldots\wedge p_n\wedge\neg q_1\wedge\ldots\wedge\neg q_k\to r\in\mathcal{P}} u_2(p_1) \wedge \ldots \wedge u_2(p_n) \wedge \neg u_1(q_1) \wedge \ldots \wedge \neg u_1(q_k).$$

Symmetric and approximation functions

Observing carefully the definitions of the functions $f_{\mathcal{P}}$ and $g_{\mathcal{P}}$ one can see that

$$f_{\mathcal{P}}(u_1, u_2) = g_{\mathcal{P}}(u_2, u_1)$$

which is again a nice enough property deserving a name:

Definition: Symmetric function.

A function $f = \langle f_1, f_2 \rangle : P \times P \to P \times P$ is symmetric if $f_1(x, y) = f_2(y, x)$ for each $x, y \in P$.

That is, if a function maps pairs into pairs, and swapping the input coordinates the output coordinates also get swapped, then the function is called symmetric.

Thus, the function $\Psi_{\mathcal{P}}$ is symmetric.

Our current aim is to show that $\Psi_{\mathcal{P}}$ is also $\leq_p$-monotone. That is, if $(u, v) \leq_p (u', v')$, then $\Psi_{\mathcal{P}}(u, v) \leq_p \Psi_{\mathcal{P}}(u', v')$. (Note that this $\leq_p$ is the precision ordering on $\mathbf{2}^Z \times \mathbf{2}^Z$, that is, $(u, v) \leq_p (u', v')$ iff $(u(q), v(q)) \leq_p (u'(q), v'(q))$ for each $q \in Z$. Technically, the precision ordering $\leq_p$ is defined on $\mathbf{4} = \mathbf{2} \times \mathbf{2}$, then it's taken pointwise on $\mathbf{4}^Z$, finally it's transformed by the isomorphism between $\mathbf{4}^Z$ and $\mathbf{2}^Z \times \mathbf{2}^Z$.)

It turns out that it's enough to show $\leq_p$-monotonicity of $f_{\mathcal{P}}$ since:

Proposition

A symmetric function $f = \langle f_1, f_2 \rangle : P^2 \to P^2$ is $\leq_p$-monotone if and only if so is f_1.

Proof

Note that f_1 is a function from $P^2 \to P$. For such functions, $\leq_p$-monotonicity means that if $(x, y) \leq_p (x', y')$, then $f_1(x, y) \leq f_1(x', y')$.

So assume f_1 is $\leq_p$-monotone and let $(x, y) \leq_p (x', y')$. Then,

$$f(x, y) = (f_1(x, y), f_2(x, y)) = (f_1(x, y), f_1(y, x)).$$

By $\leq_p$-monotonicity of f_1 and $(x, y) \leq_p (x', y')$ we get $f_1(x, y) \leq f_1(x', y')$. Also, $(x, y) \leq_p (x', y')$ means $x \leq x'$ and $y' \leq y$, thus it's also the case that $(y', x') \leq_p (y, x)$. Again by $\leq_p$-monotonicity of f_1 we get $f_1(y', x') \leq f_1(y, x)$. But then,

$$(f_1(x, y), f_1(y, x)) \leq_p (f_1(x', y'), f_1(y', x')) = (f_1(x', y'), f_2(x', y')) = f(x', y'),$$

thus f is indeed $\leq_p$-monotone.

For the other direction, assume f is $\leq_p$-monotone and $(x, y) \leq_p (x', y')$. Then

$$(f_1(x, y), f_2(x, y)) = f(x, y) \leq_p f(x', y') = (f_1(x', y'), f_2(x', y')),$$

implying $f_1(x, y) \leq f_1(x', y')$ (and also that $f_2(x', y') \leq f_2(x, y)$), that is, f_1 is $\leq_p$-monotone.

Soon we will show that $\Psi_{\mathcal{P}}$ is not only symmetric, but also a $\leq_p$-monotone function. But first, let's see why it's so good to have such a function, a so-called approximation function:

Definition: Approximation function.

A function $f : P^2 \to P^2$ is an approximation function if it is symmetric and $\leq_p$-monotone.

Approximation functions are called this way since they "approximate" some function: namely, these interval-interval functions are approximating the point-point function $x \mapsto f_1(x, x)$. To put it more precise:

Definition: Approximated function.

The $f : P^2 \to P^2$ approximation function approximates the $P \to P$ function $x \mapsto f_1(x, x)$.

Approximating means,

Proposition

If the function $f : P^2 \to P^2$ approximates the function $g : P \to P$, then $f(x, x) = (g(x), g(x))$ for each $x \in P$.

In more general, if $y \leq x \leq z$ (that is, x is contained within the interval (y, z)), then $g(x)$ is contained in $f(y, z)$.

Proof

Clearly, $f(x, x) = (f_1(x, x), f_2(x, x) = (f_1(x, x), f_1(x, x)) = (g(x), g(x))$, this holds for any symmetric function.

For the other statement, $y \leq x \leq z$ implies $(y, z) \leq_p (x, x)$, by $\leq_p$-monotonicity we get

$f(y,z) \leq_p f(x,x) = (g(x), g(x))$, that is, the point-like interval $(g(x), g(x))$ is contained in $f(y,z)$.

Thus, approximation functions map more precise inputs to more precise outputs (that's what $\leq_p$-monotonicity is stating) and output point-like intervals if the input is point-like (that's implied by symmetry).

We already know that $\Psi_{\mathcal{P}} = \langle f_{\mathcal{P}}, g_{\mathcal{P}} \rangle$ is a symmetric function. We have not shown yet it's also $\leq_p$-monotone, but nevertheless, we can ask the following: if it's an approximation function, then what function does it approximate? Well, recalling that

$$f_{\mathcal{P}}(u_1, u_2)(r) = \bigvee_{p_1 \wedge \ldots \wedge p_n \wedge \neg q_1 \wedge \ldots \wedge \neg q_k \to r \in \mathcal{P}} u_1(p_1) \wedge \ldots \wedge u_1(p_n) \wedge \neg u_2(q_1) \wedge \ldots \wedge \neg u_2(q_k)$$

we get that $\Psi_{\mathcal{P}}$ then approximates the function $u \mapsto f_{\mathcal{P}}(u,u)$, that is,

$$f_{\mathcal{P}}(u, u)(r) = \bigvee_{p_1 \wedge \ldots \wedge p_n \wedge \neg q_1 \wedge \ldots \wedge \neg q_k \to r \in \mathcal{P}} u(p_1) \wedge \ldots \wedge u(p_n) \wedge \neg u(q_1) \wedge \ldots \wedge \neg u(q_k),$$

which is familiar..., yes, it's the function $T_{\mathcal{P}}$ we started with!

So, if $\Psi_{\mathcal{P}}$ is an approximation function, then it approximates $T_{\mathcal{P}}$.

This "sounds good", since $T_{\mathcal{P}}$ is deeply related to logic programs.

$\Psi_{\mathcal{P}}$ is $\leq_p$-monotone

The problem with $T_{\mathcal{P}}$ was that it's not a monotone function, thus it offers no natural candidate for a semantics (i.e. a least pre-fixed point).

In this section we show that the function $\Psi_{\mathcal{P}}$ is monotone, with respect to the ordering $\leq_p$. Again, we will build up our proof bottom-up, breaking it into several smaller claims.

Also, we already know that it suffices to show that $f_{\mathcal{P}}$ is $\leq_p$-monotone.

We start with the literal evaluations again. In this case (since negation is involved) we only make the statement for the poset $\mathbf{2}^Z \times \mathbf{2}^Z$.

Proposition

The evaluation functions $(u_1, u_2) \mapsto u_1(p)$ and $(u_1, u_2) \mapsto \neg u_2(p)$ are $\leq_p$-monotone for each $p \in Z$.

Proof

Assume $(u_1, u_2) \leq_p (u_1', u_2')$, that is, $u_1 \leq u_1'$ and $u_2' \leq u_2$ and let $p \in Z$. Then, since the u_i are pointwise ordered, we have $u_1(p) \leq u_1'(p)$ and $u_2'(p) \leq u_2(p)$. The former implies $(u_1, u_2) \mapsto u_1(p)$ is $\leq_p$-monotone. From the latter, $u_2'(p) \leq u_2(p)$ implies $\neg u_2(p) \leq \neg u_2'(p)$, hence $(u_1, u_2) \mapsto \neg u_2(p)$ is also $\leq_p$-monotone.

Then, we again build up $f_{\mathcal{P}}$ step by step the same way as before (during the proof of the continuity of $T_{\mathcal{P}}$). But for monotonicity we don't have to deal with finite infima separately:

Proposition

If I is some index set and $f_i : P \to Q$ are monotone functions, then $\bigwedge_{i\in I} f_i$ and $\bigvee_{i\in I} f_i$, if they exist, are monotone as well.

Proof

Note that there is no need here to emphasize $\leq_p$-monotonicity: the argument works between arbitrary posets.

Let $x \leq y$. Then for each $i \in I$ we have $f_i(x) \leq f_i(y)$. Thus, any upper bound of $\{f_i(y) : i \in I\}$ is also an upper bound of $\{f_i(x) : i \in I\}$ (implying $(\bigvee f_i)(x) = \bigvee f_i(x) \leq \bigvee f_i(y) = (\bigvee f_i)(y)$) and similarly, any lower bound of $\{f_i(x) : i \in I\}$ is also a lower bound of $\{f_i(y) : i \in I\}$, implying $(\bigwedge f_i)(x) = \bigwedge f_i(x) \leq \bigwedge f_i(y) = (\bigwedge f_i)(y)$.

Of course if Q is a complete lattice, as in our case when $Q = \mathbf{2}^Z$, the supremum/infimum of such functions always exists.

Then, we have that the functions of the form

$$f_{\mathcal{P},r}(u_1, u_2) = \bigvee_{p_1\wedge\ldots\wedge p_n\wedge\neg q_1\wedge\ldots\wedge\neg q_k\to r\in\mathcal{P}} u_1(p_1) \wedge \ldots \wedge u_1(p_n) \wedge \neg u_2(q_1) \wedge \ldots \wedge \neg u_2(q_k)$$

are $\leq_p$-monotone, since the literal evaluations are monotone, evaluating the body of the clause is then an infimum of $\leq_p$-monotone functions, which is then $\leq_p$-monotone as well, and then, taking the supremum of such functions is $\leq_p$-monotone again.

Then again, $f_{\mathcal{P}}$ is the target tupling of such functions $f_{\mathcal{P},r}$, which also preserves monotonicity (and again, it does not matter that the ordering in question is $\leq_p$ or not):

Proposition

The target tupling of monotone functions is monotone.

Proof

Let I be an index set and $f_i : P \to Q$ be monotone functions for each $i \in I$. We claim that the functions $f = \langle f_i\rangle_{i\in I} : P \to Q^I$ is monotone. So let $x \leq y$ be members of P, we have to show that $f(x) \leq f(y)$. Since $f(x)$ and $f(y)$ are from Q^I, i.e., functions ordered pointwise, $f(x) \leq f(y)$ if and only if $f(x)(i) \leq f(y)(i)$ holds for each $i \in I$. But that's $f_i(x) \leq f_i(y)$ by the definition fo target tupling, which holds since each f_i is assumed to be monotone.

Thus,

Proposition

$\Psi_{\mathcal{P}}$ is $\leq_p$-monotone.

Hence, it's an approximation function since it's symmetric as well.

Well-orderings and well-founded induction

Now we know that $\Psi_{\mathcal{P}}$ is a monotone function (with respect to $\leq_p$). Our current aim is to show that monotone functions always have least (pre)fixed points (in complete lattices, that is).

In order to achieve this, we have to meet a proof method called well-founded induction, which is a generalization of induction over the naturals we've already used (in the proof of the Tarski Fixed Point Theorem, which is not a coincidence since the Kleene Fixed Point Theorem generalizes the Tarski one, by generalizing the induction method).

But first, let us see an example for a monotone function in some complete lattice. Let the poset be $P = \mathbb{R}_{\geq 0} \cup \{\infty\}$, the nonnegative real numbers equipped with a ∞ element. We have seen that it's a complete lattice.

For the function, we write each real number in the form $r = n - \alpha$, where n is an integer and $0 < \alpha \leq 1$. That is, $0.5 = 1 - 0.5$, $1.7 = 2 - 0.3$, $10.3 = 11 - 0.7$, and, for integers, $2 = 3 - 1$, $42 = 43 - 1$ and so on – n is the next strictly greater integer and α is the difference $n - r$. Then we define the function f as

$$f(n - \alpha) = n - \frac{\alpha}{2} \qquad\qquad f(\infty) = \infty$$

For example, $f(0) = 0.5$, $f(0.5) = 0.75$, $f(42.2) = 42.6$ and so on.

Let's try to produce a fixed point of f by the iteration method we already seen in the proof of the Tarski Fixed Point Theorem: start from 0, the least element of the poset, iterate f and "at the end", take the supremum:

$$x_0 = 0 \qquad x_1 = f(0) = 0.5 \qquad x_2 = f(0.5) = 0.75 \qquad x_3 = f(0.75) = 0.875$$

and so on, in general $x_n = 1 - \frac{1}{2^n}$, thus the supremum of this sequence is $\bigvee_{n\geq 0} x_n = 1$.

If f were continuous, we would have $f(1) = 1$ and we were done. However, $f(1) = 1.5$, so we are not done yet…for reasons becoming apparent later, let us refer to this element $\bigvee_{n\geq 0} x_n$ as x_ω. Then, iterating further we get

$$x_\omega = 1 \qquad x_{\omega+1} = f(1) = 1.5 \qquad x_{\omega+2} = f(1.5) = 1.75 \qquad x_{\omega+3} = f(1.75) = 1.875$$

and so on, and after another infinitely many iterations we take again a supremum and get a value, henceforth called $x_{\omega\times 2} = 2$. Then again, $x_{\omega\times 2+1} = 2.5$, $x_{\omega\times 2+2} = 2.75$, and so on, iterating infinitely many times, taking supremum puts us into $x_{\omega\times 3} = 3$ and so on.

And after infinitely many infinite iterations we arrive to $x_{\omega\times\omega} = \infty$, so we "finally" reach the unique (pre)fixed point of f.

This might seem obscure at first[4], but

it works

and it always works. Basically we just start from $\perp$, apply f and take suprema "in some structured manner" and hack our way to the least (pre)fixed point of f.

Now for this "structured manner" we have to know a little more about well-orderings and well-founded induction.

[4]for those readers who have never done that before

Definition: Well-ordering.

A strict linear order $(P, <)$ is a well-ordering if there is no infinite (strictly) descending chain

$$\ldots < x_3 < x_2 < x_1 < x_0$$

in P.

If we have a well-ordered set, then we are able to do well-founded induction:

Proposition: Well-founded induction.

Assume $(P, <)$ is a well-ordering and that $X \subseteq P$ is a set such that for any element $x \in P$, if all the elements $y < x$ strictly less than x are in X, then so is x.

Then $X = P$.

Writing the statement in a bit more formal way:

$$\forall x((\forall y(y < x \ \rightarrow \ x \in X)) \rightarrow x \in X)$$

implies $X = P$.

Proof

Assume X satisfies the above property and $X \neq P$. Then there is some $x_0 \notin X$. Since x_0 is not in X, there has to be some element $x_1 < x_0$ also not in X (since otherwise every element $y < x_0$ is in X, thus, by the condition, $x_0 \in X$ as well). Repeating this argument we get that there also exists some $x_2 < x_1$ not in X and so on, and we get an infinite descending chain, which contradicts to P being well-ordered.

In our example, we did not index our sequence $x_0, x_1, x_2, \ldots$ by only the natural numbers, but also by "something else" we called ω, $\omega + 8$, $\omega \times 2$ and so on.

It turns out that we indexed our sequence with ordinals, another set theoretic construction. (We counted up to the ordinal ω^2, which is still a fairly small ordinal – there are many more ordinals much larger than that.)

Ordinals have two nice properties why it is good to index sequences by them[5]:

- Any set of ordinals is well-ordered. (Thus, we can do well-founded induction on the sequence.)
- It does not matter how large the cardinality of a set X is, there is always an ordinal α which is "larger" than X in the following sense: the set $\{\beta : \beta < \alpha\}$ of ordinals which are less than α has a larger cardinality than X. (Thus, we can count up to pretty much anything using ordinals. Even to uncountably infinite and more.)

Ordinals

In this section we define the (von Neumann) ordinals themselves. Basically, an ordinal is a set of sets, satisfying some additional properties (which ensure that the collection of ordinals is well-ordered).

[5]assuming ZFC, for those who are interested in the math details

Definition: Ordinal.

An ordinal is a set α of sets, satisfying the following properties:

- It is transitive: if $\beta \in \alpha$ and $\gamma \in \beta$, then $\gamma \in \alpha$ as well.
- The elements of α are well-ordered with respect to the $\in$ relation.

To put the second condition more explicit: if $x \in y$ and $y \in z$ for $x, y, z \in \alpha$, then $x \in z$ (transitivity); $x \notin x$ for each $x \in \alpha$ (irreflexivity); for each $x, y \in \alpha$, exactly one of $x \in y$, $y \in x$ or $x = y$ has to hold, moreover, there is no infinite sequence $x_1, x_2, \ldots \in \alpha$ such that $\ldots \in x_3 \in x_2 \in x_1$.

In particular, if α is an ordinal, then $\alpha \notin \alpha$, since if $\alpha \in \alpha$, then α (as an element of α) violates the irreflexivity condition.

For example, the empty set $\emptyset$ is an ordinal (and is usually denoted 0). Also, $\{\emptyset\}$, the set containing the empty set is an ordinal: transitivity is OK, since $\alpha = \{\emptyset\}$, there is only one $\beta \in \alpha$, namely, $\beta = \emptyset$ and there is no $\gamma \in \emptyset$. Also, for $\{\emptyset\}$ being well-ordered it suffices to check irreflexivity, which holds since $\emptyset \notin \emptyset$. The ordinal $\{\emptyset\}$ is usually denoted 1.

Another ordinal is $\{\emptyset, \{\emptyset\}\}$ – that is, the two-element set, having the two previous ordinals 0 and 1 as elements. We could also write $\{0, 1\}$ for this ordinal, and call it 2. This ordinal 2 is transitive since $\emptyset \in \{\emptyset\} \in 2$, and $\emptyset \in 2$ as well, and its two elements are well-ordered: $\emptyset \in \{\emptyset\}$, or $0 \in 1$ if we prefer to write it this way.

Similarly, we can define $3 = \{0, 1, 2\}$, $4 = \{0, 1, 2, 3\}$ and so on – these are ordinals (and actually this is how the natural numbers are defined within set theory).

Basically, we get $n + 1$ as $n \cup \{n\}$. This works in general:

Definition: $\alpha + 1$.

If α is an ordinal, let $\alpha + 1$ denote the set $\alpha \cup \{\alpha\}$.

Proposition

If α is an ordinal, then so is $\alpha + 1$.

Proof

First we show that $\alpha + 1$ is transitive. Let $\gamma \in \beta \in \alpha + 1$. Then, either $\beta \in \alpha$ or $\beta = \alpha$. In the former case, since α is transitive, from $\gamma \in \beta \in \alpha$ we get $\gamma \in \alpha$, thus $\gamma \in \alpha \cup \{\alpha\} = \alpha + 1$ as well. In the latter case, when $\beta = \alpha$, we have $\gamma \in \beta = \alpha$, thus $\gamma \in \alpha \cup \{\alpha\} = \alpha + 1$ as well.

Now we show that $\alpha + 1$ is well-ordered by $\in$.

For irreflexivity, let $\beta \in \alpha + 1$. If $\beta \in \alpha$, then $\beta \notin \beta$ since α is well-ordered by $\in$. If $\beta = \alpha$, then we know that $\beta \notin \beta$, since α is an ordinal. Thus, $\in$ is irreflexive over $\alpha + 1$.

For transitivity, let $\beta_1, \beta_2, \beta_3 \in \alpha + 1$ be so that $\beta_1 \in \beta_2$ and $\beta_2 \in \beta_3$. If neither of them is α, then they are in α as well which is well-ordered by $\in$. If $\beta_3 = \alpha$, then we have $\beta_1 \in \beta_2 \in \alpha$ which implies $\beta_1 \in \alpha$ since α is transitive. If $\beta_2 = \alpha$ and thus $\beta_3 \neq \alpha$ (otherwise it would be the case $\alpha \in \alpha$) then we have $\beta_3 \in \alpha$, thus $\beta_3 \in \alpha$ and $\alpha \in \beta_3$ both hold which is a contradiction, since by the transitivity of α we get then $\alpha \in \alpha$. Similarly, $\beta_1 = \alpha$ also gives us a contradiction, since then $\beta_2 \in \alpha$ and $\alpha \in \beta_2$ both hold, thus $\alpha \in \alpha$ by transitivity of α.

For trichotomy, let $\beta, \gamma \in \alpha + 1$. If both of them are in α, then either $\beta \in \gamma$, $\gamma \in \beta$ or $\beta = \gamma$ since α is well-ordered. Otherwise, if $\beta = \gamma = \alpha$ then they are equal; if $\beta = \alpha$ and $\gamma \neq \alpha$, then $\gamma \in \alpha$; and if $\gamma = \alpha$ and $\beta \neq \alpha$, then $\beta \in \alpha$.

Finally, $\alpha + 1$ is well-ordered by $\in$. Assume there is an infinite descending chain $\ldots \in \beta_2 \in \beta_1 \in \beta_0$ with each β_i being a member of $\alpha + 1$. Then if $\beta_i = \alpha$, then $i = 0$, since otherwise $\alpha \in \beta_{i-1}$ which is in turn a member of $\alpha + 1$, thus it's either α or a member of α – both cases imply $\alpha \in \alpha$ which cannot happen. Thus, $\ldots \in \beta_2 \in \beta_1$ is an infinite descending chain in α which is a contradiction since α is well-ordered by $\in$.

In the previous examples all the elements of ordinals were ordinals themselves. This is not a coincidence:

Proposition

Elements of ordinals are ordinals.

Proof

Let α be an ordinal and $\beta \in \alpha$. By transitivity, all the elements of β are elements of α as well, thus they are also well-ordered by $\in$. (It is clear that any sub-ordering of a well-order is a well-order.)

We still have to show that β is transitive, so let $\delta \in \gamma \in \beta$. By α being transitive we get from $\gamma \in \beta \in \alpha$ that $\gamma \in \alpha$, which in turn implies by $\delta \in \gamma \in \alpha$ and again the transitivity of α that $\delta \in \alpha$ as well. Thus, since α is well-ordered by $\in$, and β, δ are two elements of α, either $\beta = \delta$, or $\beta \in \delta$, or $\delta \in \beta$ has to hold. Now by $\delta \in \gamma \in \beta$, each of them being a member of α which is well-ordered by $\in$, $\beta = \delta$ cannot happen (that would violate irreflexivity). Also, $\beta \in \delta$ cannot happen (applying transitivity we would get $\beta \in \beta$, violating irreflexivity again). Thus it has to be the case $\delta \in \beta$, so β is indeed a transitive set.

Also, in the above constructions for the finite ordinals we had that the "initial part" of an ordinal (say, $\{0, 1\}$ within $4 = \{0, 1, 2, 3\}$) was also an ordinal (in this case, 2). This is again not a coincidence:

Definition: Initial segment of an ordinal.

A subset X of an ordinal α is called an initial segment of α if whenever $\beta \in X$ and $\gamma \in \beta$, then also $\gamma \in X$.

That is, an initial segment is "closed downwards" with respect to $\in$. Or, equivalently, it is a transitive subset of an ordinal.

Proposition

If X is an initial segment of an ordinal, then either $X \in \alpha$ (and thus X is an ordinal), or $X = \alpha$.

Proof

First we show that X is an ordinal. X is transitive by definition. Also, since X is a subset of α and α is well-ordered by $\in$, we have that X is also well-ordered by $\in$.

Assume $X \neq \alpha$. Then there is some element $\beta \in \alpha - X$. Since $\in$ is a well-ordering on α, we have either $x \in \beta$ or $\beta \in x$ or $\beta = x$ for each $x \in X$. But since X is transitive, $\beta \in x \in X$ would imply $\beta \in X$, a contradiction; thus, since $\beta = x$ is also not an option (as $x \in X$ but $\beta \notin X$), we have $x \in \beta$ for all $x \in X$. That is, if X is a transitive subset of α, then either $X = \alpha$ or $X \subseteq \beta$ for some $\beta \in \alpha$, that is, a transitive subset of some ordinal $\beta \in \alpha$. Let us set $\beta_0 = \beta$ and for each integer $n \geq 0$, let $\beta_{n+1} \in \beta_n$ be an ordinal with $X \subseteq \beta_{n+1}$ if such an ordinal exists. Note that since $\beta_0 \in \alpha$ and $\beta_{n+1} \in \beta_n \in \alpha$ implies $\beta_{n+1} \in \alpha$ by transitivity of α, we have each β_n is a member of α. Now since α is well-ordered, there is no infinite descending chain of such β's, thus at some point we get $X = \beta_n$, thus X is indeed an element of α.

The above claim is good since it helps proving that ordinals are well-ordered:

Proposition

Any set of ordinals is well-ordered by $\in$.

Proof

We know that $\alpha \notin \alpha$ for any ordinal α, so irreflexivity is fine. Also, if $\gamma \in \beta \in \alpha$ for the ordinals α, β, γ, then by the transitivity of α we get $\gamma \in \alpha$, proving that $\in$ is transitive among ordinals.

For trichotomy, we use the previous proposition on initial segments. Let α and β be ordinals, $\alpha \neq \beta$.

Then $\gamma = \alpha \cap \beta$ is transitive, since if $\varepsilon \in \delta \in \gamma$, then by $\gamma \subseteq \alpha$ we get $\varepsilon \in \delta \in \alpha$, thus $\varepsilon \in \alpha$ by transitivity of α; and similarly by $\varepsilon \in \delta \in \beta$ we also have $\varepsilon \in \beta$, thus $\varepsilon \in \alpha \cap \beta = \gamma$, proving transitivity of γ.

Hence, by the above proposition we get that either $\gamma \in \alpha$ or $\gamma = \alpha$, and either $\gamma \in \beta$ or $\gamma = \beta$. Now assume $\gamma \in \alpha$ and $\gamma \in \beta$ both hold. Then $\gamma \in \alpha \cap \beta = \gamma$, which cannot happen since γ is an ordinal. Thus the following cases can hold:

- $\gamma \in \alpha$ and $\gamma = \beta$. In this case $\beta \in \alpha$.
- $\gamma = \alpha$ and $\gamma \in \beta$. In this case $\alpha \in \beta$.
- $\gamma = \alpha$ and $\gamma = \beta$. In this case $\alpha = \beta$.

So $\in$ is trichotome over ordinals.

Finally, assume there is an infinite descending chain $\ldots \in \alpha_2 \in \alpha_1 \in \alpha_0$ of ordinals. Then by the transitivity of α_0 we get that every α_n with $n \geq 1$ is a member of α_0: $\alpha_1 \in \alpha_0$ by definition, and applying induction we have that if $\alpha_n \in \alpha_0$, then by $\alpha_{n+1} \in \alpha_n$ and the transitivity of α_0 we get $\alpha_{n+1} \in \alpha_0$. But then α_0 is not well-ordered by $\in$, a contradiction.

From now on we will use the notation $\alpha < \beta$ instead of $\alpha \in \beta$ when it is the ordering among the ordinals that matters.

At this point it might be unclear whether there are ordinals at all that are not those of the form n for $n \geq 0$, but there are.

Proposition

Assume $X = \{\alpha_i : i \in I\}$ is a set of ordinals. Then the union $\alpha = \bigcup_{i \in I} \alpha_i$ is an ordinal, and is the supremum of this set (with respect to the well-ordering $<$ defined between ordinals).

Proof

The set X either has a largest element or not. If it has a largest element β, then for each $\gamma \in X$, $\gamma \neq \beta$ we have $\gamma < \beta$, that is, $\gamma \in \beta$ implying $\gamma \subseteq \beta$ since β is transitive. Hence in that case the union is β, clearly still an ordinal, and it is indeed the supremum of this set, being its largest element.

Now assume X does not have a largest element.

Then for each $i \in I$ we have $\alpha_i \in \alpha$ since $\alpha_i < \alpha_j$ for some $j \in I$, and α is the union of all the α_j, hence $\alpha_i \in \alpha$ as well. Thus, if α is an ordinal, then it is an upper bound of X.

To see that α is an ordinal, let $\gamma \in \beta \in \alpha$. Since α is the union of the sets α_i, $\beta \in \alpha_i$ holds for some $i \in I$, hence by transitivity of α_i we get $\gamma \in \alpha_i$, thus $\gamma \in \alpha$ as well. Thus, α is a transitive set.

To see that α is well-ordered by $\in$, we check all the properties. Assume $x, y, z \in \alpha$. Then $x \in \alpha_i$, $y \in \alpha_j$ and $z \in \alpha_k$ for some $i, j, k \in I$. Since any set of ordinals is well-ordered by $\in$, there is a greatest element in the set $\{\alpha_i, \alpha_j, \alpha_j\}$, let it be α_t. Then by transitivity we get $x, y, z \in \alpha_t$. Thus, since α_t is well-ordered by $\in$, we get that $x \notin x$, $x \in y$ and $y \in z$ imply $x \in z$ and exactly one of $x \in y$, $y \in x$ or $x = y$ holds. Thus α is strictly linearly ordered by $\in$. Now assume there is an infinite descending chain $\ldots < x_2 < x_1 < x_0$ of elements of α. Then since α is the union of the α_i sets, $x_0 \in \alpha_i$ for some $i \in I$. By transitivity of α_i we get that $x_{n+1} \in x_n \in \alpha_i$ implies $x_{n+1} \in \alpha_i$, thus in that case α_i also contains an infinite descending chain, which is a contradiction since α_i is also well-ordered by $\in$.

Hence, α is an upper bound of the ordinals α_i. Now assume β is also an upper bound. Then by $\alpha_i < \beta$ for each $i \in I$, that is, $\alpha_i \in \beta$ and β being transitive implies $\alpha_i \subseteq \beta$ for each $i \in I$. Thus, their union α is also a subset of β. Moreover, α is transitive, thus either $\alpha = \beta$, or $\alpha \in \beta$, that is, $\alpha < \beta$, thus α is indeed the supremum of the set X of ordinals.

Hence, $\omega = \{0, 1, 2, \ldots\}$ is an ordinal. (This is the same ω as the one we used previously for indexing.)

We have seen two constructions for constructing larger ordinals from smaller ones: the construction $\alpha \mapsto \alpha + 1$ and taking union of a set of ordinals. The following proposition states that these are essentially the only methods for constructing ordinals.

Proposition

Every ordinal α is either a successor ordinal, that is, an ordinal of the form $\beta + 1$, or a limit ordinal, that is, an ordinal of the form $\bigvee_{\beta < \alpha} \beta$.

Proof

We know that α is well-ordered by $\in$, which is the ordering relation among ordinals and elements of α are ordinals as well.

Now either α has a largest element with respect to $\in$, or it does not have. If β is the largest element of α, then $\alpha = \beta + 1$. Indeed, since $\beta \in \alpha$ and α is transitive, α contains all the members of β, that is, $\beta \subseteq \alpha$. Moreover, if $\gamma \in \alpha - \beta$, then either $\gamma \in \beta$, $\beta \in \gamma$ or $\beta = \gamma$; now since $\gamma \in \alpha - \beta$, the case $\gamma \in \beta$ is ruled out; since β is the largest element of α, the case $\beta \in \gamma$ is also ruled out, thus if $\gamma \in \alpha - \beta$, then $\gamma = \beta$, which means precisely that $\alpha = \beta + 1$.

Now assume α does not have a largest element with respect to $\in$. Then by transitivity, each member $\beta \in \alpha$ is a subset of α, that is, $\beta < \alpha$ implies $\beta \subseteq \alpha$, thus $\bigcup_{\beta<\alpha} \beta \subseteq \alpha$. Also, for each element $\gamma \in \alpha$ there is an ordinal $\beta \in \alpha$ with $\gamma < \beta$ (since no γ is a largest element of α), hence each element γ of α is a member of some $\beta < \alpha$, hence α itself is a subset of their union: $\alpha \subseteq \bigcup_{\beta<\alpha} \beta$, and these two statements together imply $\alpha = \bigcup_{\beta<\alpha} \beta$.

It is also clear that α is the supremum of the set $\{\beta : \beta < \alpha\}$ in this case. (Note that $\beta < \alpha$ is $\beta \in \alpha$ here, thus the latter set is α itself.)

Technically, $0 = \bigvee \emptyset$ is often not viewed as a limit ordinal but as a separate case, that is, there are three types of ordinals: zero, successor ordinals and limit (nonzero) ordinals, and these three cases are mutually exclusive.

Also, $\alpha + 1$ is denoted this way by a reason:

Proposition

Let $\alpha < \beta$ be ordinals. Then $\alpha + 1 \leq \beta$.

Proof

Since $<$ is trichotome over the ordinals, we have either $\alpha + 1 = \beta$, $\alpha + 1 < \beta$ or $\beta < \alpha + 1$. But if $\beta < \alpha + 1$, that is, $\beta \in \alpha \cup \{\alpha\}$, then either $\beta \in \alpha$ (i.e. $\beta < \alpha$) or $\beta = \alpha$, both contradicting to $\alpha < \beta$. Hence, $\alpha + 1 \leq \beta$.

Summarizing some properties of ordinals we get:

- 0 is an ordinal.
- There is a well-ordering relation $<$ over the ordinals.
- For each ordinal α, there is an ordinal denoted $\alpha + 1$.
- It holds that $\alpha < \alpha + 1$ and for each $\alpha < \beta$ we have $\alpha + 1 \leq \beta$.
- For any ordinal α, the collection of ordinals smaller than α is a set[6].
- For every ordinal α, one of the following three mutually exclusive cases hold:
  - $\alpha = 0$.

[6] Actually, this set is α itself.

- $\alpha = \beta + 1$ for some ordinal β.
- α is a nonzero limit ordinal, that is, $\alpha = \bigvee_{\beta<\alpha} \beta$.

These will be enough to show that a monotone function over a complete lattice always has a least pre-fixed point.

The Kleene Fixed Point Theorem

As the collection of the ordinals is well-ordered, we can do well-founded induction over the ordinals, e.g., defining a sequence, indexed by ordinals, within some set P, where each element is defined based on the earlier elements of the sequence.

Proposition: Kleene Fixed Point Theorem.

Let P be a complete lattice and $f : P \to P$ be a monotone function. We define to each ordinal α the following element x_α of P: $x_0 = \bot$; if $\alpha = \beta + 1$ is a successor ordinal, then $x_\alpha = f(x_\beta)$ and if $\alpha = \bigvee_{\beta<\alpha} \beta$ is a nonzero limit ordinal, then let $x_\alpha = \bigvee_{\beta<\alpha} x_\beta$.

Then, for some ordinal α, x_α is the least (pre)fixed point of f.

In the proof we will use well-founded induction.

Proof

Let $X \subseteq P$ be the set of those elements occurring in the sequence above (i.e. $x \in X$ if and only if $x = x_\alpha$ for some ordinal α). Then X contains $\bot$, since $x_0 = \bot$, and if $x \in X$, then $f(x) \in X$ as well (since if $x = x_\alpha$, then $f(x) = x_{\alpha+1}$ by definition).

Also, each x_α is a post-fixed point of f, which can be shown via well-founded induction. For $\alpha = 0$ we have $x_\alpha = \bot$ which is a post-fixed point. For successor ordinals $\alpha = \beta + 1$, if x_β is a post-fixed point, then $x_\alpha = f(x_\beta)$ is also a post-fixed point since monotone functions transform post-fixed points into post-fixed points. For limit non-zero ordinals $\alpha = \bigvee_{\beta<\alpha} \beta$, we have that a supremum of post-fixed points is still a post-fixed point. To see that, let $Y \subseteq P$ be a set of post-fixed points. Then for each $y \in Y$ we have $y \leq \bigvee Y$, thus by monotonicity we get $y \leq f(y) \leq f(\bigvee Y)$, that is, $f(\bigvee Y)$ is an upper bound for Y while $\bigvee Y$ is the least upper bound, so $\bigvee Y \leq f(\bigvee Y)$, the supremum is also a post-fixed point.

We also claim that $\bigvee X \in X$, that is, X has a greatest element. To see this, let us fix to each $x \in X$ an ordinal $\alpha(x)$ with $x = x_{\alpha(x)}$ and let α be the supremum of these ordinals (as it's the supremum of a set of ordinals, α is also an ordinal). To circumvent a case analysis, let γ be a limit ordinal larger than α (such ordinal exists, e.g. the supremum of the ordinals α, $\alpha+1$, $\alpha+2, \ldots$) Then $x_\gamma = \bigvee_{\beta<\gamma} x_\beta$ and since each $x \in X$ appears as $x = x_{\alpha(x)}$ for some $\alpha(x) < \gamma$, we get that $\{x_\beta : \beta < \gamma\}$ is X. Hence, $x_\gamma = \bigvee X$ and since γ is an ordinal, $\bigvee X \in X$, that is, X has a largest element $x = x_\gamma$.

But then x_γ is still a post-fixed point of f, thus $x_\gamma \leq f(x_\gamma) = x_{\gamma+1}$ which is also in X since $\gamma + 1$ is also an ordinal. But since x_γ is the largest element of X, we have that $x_{\gamma+1} \leq x_\gamma$ as well, thus $x_\gamma = x_{\gamma+1} = f(x_\gamma)$, hence x_γ is a fixed point of f.

It remains to show that x_γ is the least pre-fixed point. We show by well-founded induction that whenever x is a pre-fixed point of f, and α is an ordinal, then $x_\alpha \leq x$.

For $\alpha = 0$ the claim holds, since $x_0 = \bot \leq x$ for any $x \in P$.

If $\alpha = \beta + 1$ is a successor ordinal, then by induction we have $x_\beta \leq x$. By monotonicity of f we get $x_\alpha = x_{\beta+1} = f(x_\beta) \leq f(x)$ but since x is a pre-fixed point, we also have $f(x) \leq x$, implying $x_\alpha \leq x$.

Finally, if $\alpha = \bigvee_{\beta<\alpha} \beta$ is a limit ordinal, then by the induction hypothesis each x_β with $\beta < \alpha$ is a lower bound of x, that is, x is an upper bound of the set $\{x_\beta : \beta < \alpha\}$. Since $x_\alpha = \bigvee_{\beta<\alpha} x_\beta$ is the least upper bound, we get $x_\alpha \leq x$ also in this case.

Hence, x_γ is the least (pre)fixed point of f for some ordinal γ. (And for each ordinal $\delta > \gamma$, $x_\delta = x_\gamma$ from that point.)

Thus, the function Ψ_P always have a least (pre)fixed point with respect to the ordering $\leq_p$. This fixed point is called the Kripke-Kleene semantics of P.

The Kripke-Kleene semantics is consistent

We have seen that Ψ_P is an approximation function (that is, $\leq_p$-monotone and symmetric). Such functions always have a least (pre)fixed point due to the Kleene theorem. In this part we show that this fixed point is consistent.

First, we show that the Kleene iteration produces a linearly ordered set.

Proposition

Let P be a poset with a least element $\bot$ and $f : P \to P$ a monotone function such that the sequence

- $x_\alpha = \bot$ for $\alpha = 0$,
- $x_\alpha = f(x_\beta)$ for successor ordinals $\alpha = \beta + 1$,
- $x_\alpha = \bigvee_{\beta<\alpha} x_\beta$ for limit ordinals $\alpha = \bigvee_{\beta<\alpha} \beta$

is well-defined for each ordinal α.

Then if $\beta < \alpha$, then $x_\beta \leq x_\alpha$, thus the set $\{x_\alpha \in P : \alpha \text{ is an ordinal}\}$ is a linearly ordered subset of P.

Proof

Let α be an ordinal. We apply well-founded induction on α So assume the claim holds for each ordinal less than α, that is, whenever $\gamma < \delta < \alpha$, then $x_\gamma \leq x_\delta$.

We distinguish three cases according to whether α is 0, a successor or a limit ordinal.

Suppose $\alpha = 0$. Then there is no ordinal $\beta < \alpha$, so the claim is vacuously satisfied.

Suppose $\alpha = \beta + 1$ is a successor ordinal and let $\gamma < \alpha$ be some smaller ordinal. Then either $\gamma = \beta$ or $\gamma < \beta$ since there are no ordinals between β and $\alpha = \beta + 1$. If $\gamma = \beta$, then $x_\beta \leq f(x_\beta) = x_\alpha$, since each member of the sequence is a post-fixed point of f. If $\gamma < \beta$, then applying the induction hypothesis on $\gamma < \beta\alpha$ we get $x_\gamma \leq x_\beta$, thus by $x_\beta \leq x_\alpha$ we get $x_\gamma \leq x_\alpha$ as well.

Finally, suppose $\alpha = \bigvee_{\beta<\alpha} \beta$ is a limit ordinal. Then for each $\gamma < \alpha$ we have $x_\gamma \leq \bigvee_{\beta<\alpha} x_\beta = x_\alpha$ since x_γ is a member of the set $\{x_\beta : \beta < \alpha\}$ and x_α is an upper bound of this set.

If the ordering in question is $\leq_p$ on P^2 for some complete lattice P, then the following proposition is helpful:

Proposition

Assume $U \subseteq P^2$ is a set of consistent pairs, linearly ordered with respect to $\leq_p$. Then $\bigoplus U$ is consistent as well.

Equivalently, one can read the above proposition as "the consistent pairs in P^2 form a complete poset". Not a complete lattice, though, as e.g. $\bigoplus\{(0,0),(1,1)\}$ is $(0 \vee 1, 0 \wedge 1) = (1,0)$ which is inconsistent. But this is due to the fact that $(0,0)$ and $(1,1)$ are incomparable with respect to $\leq_p$.

Proof

Let $U = \{(x_i, y_i) : i \in I\}$ be a linearly ordered subset of P^2 (with respect to $\leq_p$). Let X denote the set $\{x_i : i \in I\}$ and Y denote $\{y_i : i \in I\}$. Then $\bigoplus U = \left(\bigvee X, \bigwedge Y\right)$. We have to show $\bigvee X \leq \bigwedge Y$, which is implied by $X \leq Y$. So let $x_i \in X$ and $y_j \in Y$, we show $x_i \leq y_j$. Since U is linearly ordered, we either have $(x_i, y_i) \leq_p (x_j, y_j)$ or the other way around.

If $(x_i, y_i) \leq_p (x_j, y_j)$, then $x_i \leq x_j$ (by $\leq_p$) and $x_j \leq y_j$ (since (x_j, y_j) is consistent), thus $x_i \leq y_j$.

If $(x_j, y_j) \leq_p (x_i, y_i)$, then $x_i \leq y_i$ (by consistency of (x_i, y_i)) and $y_i \leq y_j$ (by $\leq_p$), thus $x_i \leq y_j$.

Hence $X \leq Y$, thus indeed $\bigvee X \leq \bigwedge Y$, hence $\bigoplus U$ is indeed consistent.

Now we are ready to show consistency of the $\leq_p$-least (pre)fixed point of an approximation function:

Proposition

Assume f is an approximation function on P^2 for a complete poset P. Then the least (pre)fixed point of f (with respect to $\leq_p$) is consistent.

Proof

We show that each member x_α of the Kleene iteration sequence is consistent. Since the least (pre)fixed point of f also has this form, it is consistent as well. We apply well-founded induction on α.

For $\alpha = 0$, the claim holds since in that case, $x_\alpha = (\bot, \top)$ is the $\leq_p$-minimal element of P^2 which is consistent since $\bot \leq \top$.

For successor ordinals $\alpha = \beta + 1$, we have that $x_\beta = (x_\beta^1, x_\beta^2)$ is consistent by the induction hypothesis. Then,

$$f(x_\beta) = (f_1(x_\beta^1, x_\beta^2), f_2(x_\beta^1, x_\beta^2)) = (f_1(x_\beta^1, x_\beta^2), f_1(x_\beta^2, x_\beta^1))$$

by symmetry of f. Now since f is an approximation function, f_1 is $\leq_p$-monotone and from $x^1_\beta \leq x^2\beta$ we get that $(x^1_\beta, x^2_\beta) \leq_p (x^2_\beta, x^1_\beta)$, thus $f_1(x^1_\beta, x^2_\beta) \leq f_1(x^2_\beta, x^1_\beta)$ hence x_α is consistent as well.

For limit ordinals $\alpha = \bigvee_{\beta<\alpha} \beta$ we have $x_\alpha = \bigoplus_{\beta<\alpha} x_\beta$. Applying the induction hypothesis we get that each x_β is consistent, hence x_α is a $\leq_p$-supremum of consistent pairs, hence it is consistent as well by the previous proposition.

Thus, summarizing the results so far on generalized logic programs:

Given a generalized logic program P, we have so far

- defined a function $\Phi_P : \mathbf{4}^Z \to \mathbf{4}^Z$, similar to T_P but using 4-valued logic;
- rewritten this function to $\Psi_P : \mathbf{2}^Z \times \mathbf{2}^Z \to \mathbf{2}^Z \times \mathbf{2}^Z$, which is mathematically easier to handle but still the same function (modulo isomorphism on the domain);
- have shown this function to be an approximation function, that is, $\leq_p$-monotone and symmetric;
- have shown that monotone functions always have a least (pre)fixed point on a complete poset;
- have shown that the least (pre)fixed point with respect to $\leq_p$ of any approximation function is consistent.

Thus, the Kripke-Kleene semantics can give us a unique model. But there are some problems with it.

An example

Let us consider the following generalized (first-order) logic program:

$$\to p \qquad \neg p \to q(0) \qquad q(x) \to q(s(x)) \qquad q(x) \to r \qquad r \to s \qquad r \wedge s \to t$$

As the ground terms are 0, $s(0)$, $s(s(0))$, etc, writing q_n in place of $q(s^n(0))$ we get the Herbrand extension

$$\begin{array}{cccccc}
\to p & \neg p \to q_0 & q_0 \to q_1 & q_0 \to r & r \to s & r \wedge s \to t \\
 & & q_1 \to q_2 & q_1 \to r & & \\
 & & q_2 \to q_3 & q_2 \to r & & \\
 & & q_3 \to q_4 & q_3 \to r & &
\end{array}$$

and so on. Writing out Φ_P explicitly for each variable we get

$$\begin{aligned} \Phi_P(u)(p) &= \mathsf{t} \\ \Phi_P(u)(q_0) &= \neg u(p) \\ \Phi_P(u)(q_{n+1}) &= u(q_n) \\ \Phi_P(u)(r) &= \bigvee_{n\geq 0} u(q_n) \\ \Phi_P(u)(s) &= u(r) \\ \Phi_P(u)(t) &= u(r) \wedge u(s). \end{aligned}$$

Then, moving on with the Kleene iteration, we have to start from the $\leq_p$-least element, that is, $u_0(z) = \perp$ for each variable $z \in Z$. Then, during the iteration steps we get

step	p	q_0	q_1	q_2	q_3	q_4	q_5	$\dots$	r	s	t
0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\dots$	$\perp$	$\perp$	$\perp$
1	t	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\dots$	$\perp$	$\perp$	$\perp$
2	t	f	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\dots$	$\perp$	$\perp$	$\perp$
3	t	f	f	$\perp$	$\perp$	$\perp$	$\perp$	$\dots$	$\perp$	$\perp$	$\perp$
4	t	f	f	f	$\perp$	$\perp$	$\perp$	$\dots$	$\perp$	$\perp$	$\perp$
5	t	f	f	f	f	$\perp$	$\perp$	$\dots$	$\perp$	$\perp$	$\perp$
6	t	f	f	f	f	f	$\perp$	$\dots$	$\perp$	$\perp$	$\perp$
n	t	$q_0, \dots, q_{n-2}$ are f, the rest is $\perp$							$\perp$	$\perp$	$\perp$
ω	t	f	f	f	f	f	f	$\dots$	$\perp$	$\perp$	$\perp$
$\omega+1$	t	f	f	f	f	f	f	$\dots$	f	$\perp$	$\perp$
$\omega+2$	t	f	f	f	f	f	f	$\dots$	f	f	f

which is a fixed point of Φ_P, thus the Kripke-Kleene semantics for this particular program P sets p to t, and everything else to f (and $\omega + 2$ steps are actually needed in order to reach this particular fixed point via iteration).

Problems with the Kripke-Kleene semantics

Let us consider the following (negation-free) program P:

$$\to p \qquad p \to q \qquad p \wedge q \to r \qquad p \wedge s \to t \qquad u \to t$$

Then, the Kleene iteration on P goes as

step	p	q	r	s	t	u
0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
1	t	$\perp$	$\perp$	$\perp$	$\perp$	f
2	t	t	$\perp$	$\perp$	$\perp$	f
3	t	t	t	$\perp$	$\perp$	f
4	t	t	t	$\perp$	$\perp$	f

that is, we reach the least fixed point in step 3, while the canonical semantics, that is, iterating T_P goes as

step	p	q	r	s	t	u
0	f	f	f	f	f	f
1	t	f	f	f	f	f
2	t	t	f	f	f	f
3	t	t	t	f	f	f
4	t	t	t	f	f	f

(if we identify 0 with $(0,0) = \mathsf{f}$ and 1 with $(1,1) = \mathsf{t}$ when iterating T_P).

That is, if u_K denotes the Kripke-Kleene semantics of P and u_C denotes the canonical semantics of P, then we have $u_K \neq u_C$! More precisely, we have $u_K \leq_p u_C$ and $u_C \leq_t u_K$. In other words:

- The Kripke-Kleene semantics is not necessarily the same as the canonical semantics for (negation-free) logic programs.
- The Kripke-Kleene semantics does not always minimize the truth values.

Since we already argued that the canonical semantics is "the" good semantics of negation-free logic programs, and also, our first rule is that we want to have a $\leq_t$-minimal model as semantics, we have to modify the Kripke-Kleene semantics to suit our needs better.

Stabilizer functions and well-founded fixed points

When $f : P \to P$ is a monotone function on a complete poset P, let $\mu x.f(x)$ denote the least fixed point of f.

Thus, if $f_1 : P^2 \to P$ is a function such that for each $y \in P$, the function $f_1^y : P \to P$ defined as $f_1^y(x) := f_1(x,y)$, is a monotone function, then the $\mu x.f_1(x,y)$ is a $P \to P$ function which assigns to a value y the least such x^* with $f_1(x^*,y) = x^*$.

For example, if $f_1(x,y) = \frac{5x-y}{2}$ is an $\mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ function, then $\mu x.f_1(x,y)$ is the function $y \mapsto \frac{y}{3}$, since $\frac{y}{3}$ is the least value z satisfying $z = \frac{5z-y}{2}$. (Actually, it's the only such value.)

Using the μ notation we can define the stabilizer function of an approximation function:

Definition: Stabilizer function

Let P be a complete lattice and $f = \langle f_1, f_2 \rangle : P^2 \to P^2$ an approximation function. The stabilizer function of f is the function $s : P^2 \to P^2$ defined as $s(x,y) = (s_1(y), s_1(x))$, where $s_1 : P \to P$ is the function $s_1(z) = \mu x.f_1(x,z)$.

So e.g., $s_1(y)$ is the least fixed point of the function $z \mapsto f_1(z,y)$. This least fixed point exists, since if f is an approximation function, then f_1 is $\leq_p$-monotone, thus if $z_1 \leq z_2$, then from $(z_1,y) \leq_p (z_2,y)$ we get $f_1(z_1,y) \leq_p f_1(z_2,y)$, thus $z \mapsto f_1(z,y)$ is a monotone function and hence has a least fixed point.

The first important property of stabilizer functions is that they are still approximation functions:

Proposition

If $s : P^2 \to P^2$ is the stabilizer function of the approximation function f, then s is also an approximation function.

Proof

First, since $s(y, x) = (s_1(y), s_1(x))$, swapping the input arguments of s results in swapping the output arguments, thus s is symmetric.

To see that s is also $\leq_p$-monotone, first note that if $y \leq y'$, then $s_1(y') \leq s_1(y)$. Indeed, let x denote $s_1(y)$. Then, by definition of $s_1(y)$, we have $x = f_1(x, y)$. By $y \leq y'$ we have $(x, y') \leq_p (x, y)$, thus applying $\leq_p$-monotonicity of f_1 we get $f_1(x, y') \leq_p f_1(x, y) = x$, thus x is a pre-fixed point of the function $z \mapsto f_1(z, y')$. Since by definition of s_1, the value $s_1(y')$ is the least (pre-)fixed point of this function, we get that $s_1(y') \leq x = s_1(y)$.

Applying this we get that if $(x, y) \leq_p (x', y')$, then $x \leq x'$, implying $s_1(x') \leq s_1(x)$ and $y' \leq y$, implying $s_1(y) \leq s_1(y')$, thus $s(x, y) = (s_1(y), s_1(x)) \leq_p (s_1(y'), s_1(x')) = s(x', y')$, hence s is $\leq_p$-monotone.

The second important property of stabilizer functions is that their fixed points are "good" fixed points of the original approximation function:

Proposition

If s is the stabilizer function of f, then each fixed point of s is a $\leq_t$-minimal fixed point of f.

Proof

Let (x, y) be a fixed point of s. Then, $(x, y) = s(x, y) = (s_1(y), s_1(x))$, that is, $x = s_1(y)$ and $y = s_1(x)$. From $x = s_1(y)$ we get $x = f_1(x, y)$ and from $y = s_1(x)$ we get $y = f_1(y, x)$, thus $f(x, y) = (f_1(x, y), f_1(y, x)) = (x, y)$ holds and (x, y) is also a fixed point of f.

For $\leq_t$-minimality, let $(x', y') \leq_t (x, y)$ be a fixed point of f. That is, $x' \leq x$, $y' \leq y$ and $f(x', y') = (x', y')$, that is, $x' = f_1(x', y')$ and $y' = f_1(y', x')$. Since f_1 is $\leq_p$-monotone, we have $(y', x) \leq_p (y', x')$, implying $f_1(y', x) \leq_p f_1(y', x') = y'$, thus y' is a pre-fixed point of the function $z \mapsto f_1(z, x)$, and by definition of s_1, y is the least pre-fixed point of this function, thus $y \leq y'$, hence $y = y'$.

Analogously, $(x', y) \leq_p (x', y')$ implies $f_1(x', y) \leq f_1(x', y') = x'$, thus x' is a pre-fixed point of $z \mapsto f_1(z, y)$, while x is its least pre-fixed point, thus $x \leq x'$, hence $x = x'$ as well.

So the fixed points of the stabilizer functions are "good", since these are also fixed points of the original approximation function, and they are $\leq_t$-minimal. Thus these fixed points also deserve a name:

Let f be an approximation function and s be its stabilizer function. Fixed points of s are called stable fixed points of f.

Since s is also an approximation function, it is also $\leq_p$-monotone, so it has a least fixed point with respect to $\leq_p$. Thus, f has a $\leq_p$-least stable fixed point. That's the semantics we seek for!

Definition: Well-founded fixed point

The well-founded fixed point of an approximation function f is its $\leq_p$-least stable fixed point.

Now if f is an approximation function, $k(f)$ is its Kripke-Kleene fixed point, and $w(f)$ is its well-founded fixed point, then (since $k(f)$ is the $\leq_p$-least fixed point of f) we have $k(f) \leq_p w(f)$, so the well-founded fixed point is always at least as precise as the Kripke-Kleene fixed point. Moreover, since s is also an approximation function, and $w(f)$ is its Kripke-Kleene fixed point (that is, $w(f) = k(s)$), which is always consistent, the well-founded fixed point is also consistent. Thus we have:

The well-founded fixed point of an approximation function is always consistent, $\leq_t$-minimal, and at least as precise as its Kripke-Kleene fixed point.

The well-founded semantics coincides with the canonical semantics

We have already seen that Ψ_P approximates T_P for any (generalized) logic program P.

Now if P is negation-free, then

$$f_P(u, v)(r) = \bigvee_{p_1 \wedge \ldots \wedge p_n \to r \in P} u(p_1) \wedge \ldots \wedge u(p_n).$$

Thus, f_P does not depend on v! Hence, the function $s_1(v) = \mu u. f_P(u, v)$ is computing the least u^* with $u^* = f_P(u^*, v)$ where

$$f_P(u^*, v)(r) = \bigvee_{p_1 \wedge \ldots \wedge p_n \to r \in P} u^*(p_1) \wedge \ldots \wedge u^*(p_n) = T_P(u^*),$$

that is, u^* is the least fixed point of T_P.

In other words, $s_1(v)$ is the canonical semantics of P for every v, thus $s(x, y) = (s_1(y), s_1(x))$ is a constant function: $s(x, y) = (u, u)$ where u is the canonical semantics of P, hence its only (and thus least) fixed point is (u, u), the canonical semantics of P. (if we identify point-like intervals (x, x) with the value x).

Hence we have shown that

Proposition

The well-founded semantics of a negation-free logic program coincides with its canonical semantics.

Thus, the well-founded semantics of generalized logic programs is always a $\leq_t$-minimal fixed point, which coincides with the canonical semantics of negation-free logic programs.

Semantics of functional programs

In the second part of the course, we give a semantics for (pure) functional programs.

For those readers who have not yet seen functional programs at all, these are called functional because not only objects (coming from some typed universe, like strings or integers) but also functions are "first-class citizens" of the language. That is, if (say) int is a type and string is a type, then one can define a function of type $\mathsf{string} \to \mathsf{int}$ (that take as input a string and produce an integer, like the StrLen function which computes the length of a given string), as usual in imperative languages; but atop of that, one can also define functions of type $(\mathsf{string} \to \mathsf{int}) \to \mathsf{string}$ (that take as input a $\mathsf{string} \to \mathsf{int}$ function and produces a string as output), or taking even further, $(\mathsf{string} \to \mathsf{int}) \to (\mathsf{string} \to \mathsf{int})$ functions: these take a $\mathsf{string} \to \mathsf{int}$ function and produce another $\mathsf{string} \to \mathsf{int}$ function. For example, the "double the output value of the given function" can be defined in Scala as:

$$\mathsf{def}\ \mathrm{double}(f : \mathsf{String} \Rightarrow \mathsf{Int}) \ = \ (s : \mathsf{String}) \Rightarrow \{2 * f(s)\}$$

Then, double(strlen) becomes a $\mathsf{String} \to \mathsf{Int}$ function, returning twice the length of the input string.

Syntax: Types

Every programming language has built-in types, or base types. The set of these types will be denoted $\mathsf{BaseTypes}$. In our example language there is only one type, nat (intended to be the type of natural numbers), that is, $\mathsf{BaseTypes} = \{\mathsf{nat}\}$. For other languages we might have $\mathsf{BaseTypes} = \{\mathsf{int}, \mathsf{String}, \mathsf{bool}, \mathsf{double}\}$ or something similar, the underlying theory is the same.

From the base types, one can build up functional types as in the above example, e.g. a function of type $(\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat}$ takes a $\mathsf{nat} \to \mathsf{nat}$ function as input and produces a single nat (a number) as output, an example for such a function is $\textsc{EvalAtZero}(f) = f(0)$, that is, the function EvalAtZero gets a function f and returns the value of f for the number 0. Now a function of type $\mathsf{nat} \to (\mathsf{nat} \to \mathsf{nat})$ takes a number n as input and produces a $\mathsf{nat} \to \mathsf{nat}$ function, an example for such a function is $\textsc{AddConstant}(n) = x \Rightarrow x + n$, that is, for an input n the function $x \mapsto x + n$ is returned.

Thus, parentheses are important in the following definition:

Definition: Types

For a fixed nonempty set $\mathsf{BaseTypes}$ of base types, the set Types of types is the least set such that

i) every base type is a type,

ii) whenever σ and τ are types, then so is $(\sigma \to \tau)$. (These types are also called functional types).

Types are usually denoted by σ and τ. In order to ease notation we will write $\sigma_1 \to \sigma_2 \to \sigma_3$ in place of $\sigma_1 \to (\sigma_2 \to \sigma_3)$ sometimes, and we will omit outermost parentheses most of the time. For example, the above function AddConstant has type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$.

In our proofs we will move from more complex function types towards the base types; we can do that since the types are well-ordered:

Proposition

Any type τ can be written uniquely as $\tau = \tau_1 \to \tau_2 \to \ldots \tau_n$ with $n \geq 1$ and τ_n being a base type.

Proof

If τ is a base type, then choosing $\tau = \tau_1$, $n = 1$ is fine.

Otherwise, $\tau = \tau_0 \to \sigma$ is a functional type. Then by induction, σ uniquely can be written as $\sigma = \tau_1 \to \tau_2 \to \ldots \to \tau_n$ with τ_n being a base type, hence writing $\tau = \tau_0 \to \tau_1 \to \tau_2 \to \ldots \to \tau_n$ is fine.

We'll write programs, or terms, which will have variables. Variables will have types. The type declaration of these variables is called a context:

Definition

A context is a sequence $x_1 : \tau_1, \ldots, x_n : \tau_n$, where the x_i are pairwise different variables (coming from some fixed set $\mathcal{X}$ of variables), and the τ_i are types.

Contexts are usually denoted by Γ or Δ.

So in our example language, a program is called a term and consists of three parts: i) a variable declaration part – that is, a context, ii) the "code of the program" itself, this part is called a pure term, and iii) the type of the output of the program. So, a term is written as

$$\Gamma \ \vdash \ M \ : \ \sigma$$

where Γ is a context (declaring the types of the "global" variables of M), M is a pure term (defined below) and σ is a type. Such a tuple intuitively can be read as "if the variables of M have the types given in Γ, then the output of M will have type σ".

Functional Programming example: Ackermann in Scala

Before proceeding, let us have a look at the central notions of functional programming in the language Scala. A fine example to implement is the Ackermann function defined as

$$A(n,m) = \begin{cases} m+1 & \text{if } n = 0; \\ A(n-1,1) & \text{if } n > 0 \text{ and } m = 0; \\ A(n-1, A(n, m-1)) & \text{if } n, m > 0. \end{cases}$$

A sample implementation[7] of the above function in Scala looks like

```
def ack( n : Int, m : Int ) : Int = {
  if ( n == 0 ) m + 1
  else if ( m == 0 ) ack( n - 1 , 1 )
  else ack( n - 1 , ack( n , m - 1 ) )
}
```

[7] The sample codes of this section are available on Pastebin here and can be played around with online here.

which is absolutely fine. However, for demostration purposes we now rewrite the above code into a more "cryptic" form, with functions taking and returning functions as well.

Observe that for $n > 0$ we have

$$\begin{aligned} A(n,m) &= A(n-1, A(n,m-1)) = A(n-1, A(n-1, A(n,m-2))) \\ &= A(n-1, A(n-1, A(n-1, A(n,m-3)))) = \dots \\ &= A(n-1, A(n-1, A(n-1, \dots, A(n-1,1)))) \end{aligned}$$

with the number of applications of the function $A(n-1, \_)$ being $m+1$. That is, the function $m \mapsto A(n,m)$ can be computed as: i) take the function $x \mapsto A(n-1,x)$ and ii) iterate this function $m+1$ times, starting from 1.

For iteration, we might come up with the following idea:

```
def iterate( f : Int=>Int, n : Int ) : Int = {
  if( n == 0 ) f(1)
  else f( iterate( f , n - 1 ))
}
```

That is, the `iterate` function takes a function f and an integer n, and computes $f^{n+1}(1)$ in a recursive manner.

This is not bad for a start, but when we have a function of the form $A \times B \to C$ for the types A, B and C, we can always rewrite it to a function of the form $A \to (B \to C)$. That is, for a function $f : A \times B \to C$, instead of taking both arguments at the same time, we only take the first argument in the first place, say $a \in A$, and produce a function $f_a : B \to C$. This function plainly "remembers" the value of $a \in A$, and takes $b \in B$ as argument, then returns $f_a(b) := f(a,b)$. One can see this method as "partially assigning values to some of the arguments".

This operation in general is called currying (named after Haskell Brooks Curry among other things) and is the reason why we only take functions of the form $\sigma \to \tau$ but not function of the form $(\sigma_1 \times \sigma_2 \times \dots \sigma_n) \to \tau$, since such a function can always be curried into of type $\sigma_1 \to (\sigma_2 \to (\dots \to (\sigma_n \to \tau)))$.

The curried version of `iterate` looks like this:

```
def iterateCurried( f : Int=>Int ) : Int=>Int = {
  n => if( n == 0 ) f(1) else f( iterateCurried(f)(n-1))
}
```

Thus, the type of the function is $(\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int})$, since the uncurried version had the type $((\mathsf{Int} \to \mathsf{Int}) \times \mathsf{Int}) \to \mathsf{Int}$. From the implementation point of view, we essentially have to write the same code (apart from slight syntactical changes, like replacing the return type and beginning the body with `n=>` since the return type is now a function, not a base type). But, if we iterate the function $x \mapsto x+1$ for a couple of numbers, as $5, 7$ and 9, then using the uncurried variant we have to code

```
iterate( _+1, 5 )
iterate( _+1, 7 )
iterate( _+1, 9 )
```

that is, repeating parts of the code while using the curried variant we might code as

```
def myIter = iterateCurried( _+1 )
myIter( 5 )
myIter( 7 )
myIter( 9 )
```

which is a bit less error-prone thing to do. Also, if one changes her/his mind and wants to iterate some other function instead, then the code has to be updated only at one point which is also a good thing.

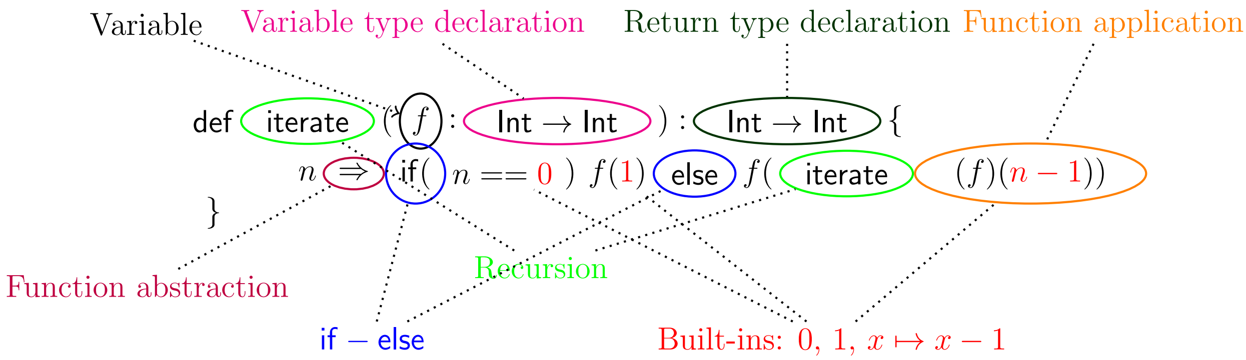
Since we have the iterating function, we can finish the second[8] implementation of the Ackermann function as

```
def ackCurried( n : Int ) : Int => Int = {
  if( n == 0 ) _ + 1
  else iterateCurried( ackCurried( n - 1 ) )
}
```

and one can play around with the values `ackCurried(2)(2)=7`, `ackCurried(3)(3)=61` and with pretty much any value over 4 (for the first argument) throwing a stack overflow exception but that's OK, the Ackermann function is an extremely fast growing function despite its mild appearance.

Syntax: λ-terms

Let us collect the fundamental notions we used, renaming the functions we implemented.



The mathematical framework for modeling functional languages is called (typed) λ-calculus. (In the untyped λ-calculus there are no types.) One can rewrite a program, like the above Scala snippet, into a so-called λ-term. Such a term consists of three parts: a context Γ, a "code" part M (that will be called a pure term later on) and an "output" type σ, and is written as

$$\Gamma \vdash M : \sigma$$

which can be understood as "if the variables of M have the types as given in Γ, then M is a valid λ-expression having type σ", intuitively.

There are strict rules according to which λ-terms can be constructed. Such a rule can be written as a formal fraction $\frac{\Gamma_1 \vdash M_1 : \sigma_1, \ldots, \Gamma_n \vdash M_n : \sigma_n}{\Gamma \vdash M : \sigma}$ and has the meaning "if each $\Gamma_i \vdash M_i : \sigma_i$ is a λ-term, then so is $\Gamma \vdash M : \sigma$".

[8] or third if you consider `iterate` as a separate attempt

A basic rule is that of **variable evaluation**, written as

$$\overline{\Gamma_1, x : \sigma, \Gamma_2 \ \vdash \ x \ : \ \sigma}$$

that is, if a context Γ declares a variable x with type σ, then x is an expression with type σ. (That's what contexts are for, after all.)

For examples, $x : \mathsf{nat}, f : \mathsf{nat} \to \mathsf{nat} \ \vdash \ x \ : \ \mathsf{nat}$ is a valid λ-term: if x is a variable of type nat and f is a variable of type $\mathsf{nat} \to \mathsf{nat}$, then (surprise) x is an expression of type nat.

The second rule, or rather a set of rules, is that of **built-ins**:

$$\frac{}{\Gamma \ \vdash \ 0 \ : \ \mathsf{nat}} \qquad \frac{}{\Gamma \ \vdash \ 1 \ : \ \mathsf{nat}} \qquad \frac{\Gamma \ \vdash \ M \ : \ \mathsf{nat}}{\Gamma \ \vdash \ \mathsf{pred}(M) \ : \ \mathsf{nat}} \qquad \frac{\Gamma \ \vdash \ M \ : \ \mathsf{nat}}{\Gamma \ \vdash \ \mathsf{succ}(M) \ : \ \mathsf{nat}}$$

That is: in any context, 0 and 1 are valid expressions of type nat, and if in some context M is an expression of type nat, then so are $\mathsf{pred}(M)$ and $\mathsf{succ}(M)$ (both having also type nat). In the standard semantics, nat will be interpreted by the set $\mathbb{N}$ of natural numbers, 0 and 1 will be interpreted by 0 and 1, respectively, succ will be the $n \mapsto n+1$ function, and pred will be the $n \mapsto n-1$ function (with the exception that $\mathsf{pred}(0) = 0$, in order to stay within the set of naturals).

For examples, it holds that $\vdash \ 0 \ : \ \mathsf{nat}$, $x : \mathsf{nat} \ \vdash \ 1 \ : \ \mathsf{nat}$, $x : \mathsf{nat} \ \vdash \ \mathsf{succ}(\mathsf{succ}(x)) \ : \ \mathsf{nat}$ and $\vdash \ \mathsf{succ}(\mathsf{pred}(1)) \ : \ \mathsf{nat}$.

The next rule is that of **if-else**. In (this variant of the) λ-calculus, one can test whether an expression evaluates to zero and return a natural number based on the condition. That is,

$$\frac{\Gamma \ \vdash \ M_1 \ : \ \mathsf{nat}, \quad \Gamma \ \vdash \ M_2 \ : \ \mathsf{nat}, \quad \Gamma \ \vdash \ M_3 \ : \ \mathsf{nat}}{\Gamma \ \vdash \ \mathsf{ifzero}(M_1, M_2, M_3) \ : \ \mathsf{nat}}$$

having the intuitive meaning "if in a context, M_1, M_2 and M_3 are all expressions having type nat, then the $\mathsf{ifzero}(M_1, M_2, M_3)$ is also an expression of type nat.

The semantics of such an expression would be written in Scala as `if(M1==0) M2 else M3`, that is, one evaluates M_1, and if its value is 0, then this term has the value of M_2, otherwise the value of M_3.

For example, $x : \mathsf{nat}, y : \mathsf{nat} \ \vdash \ \mathsf{ifzero}(x, 0, \mathsf{succ}(y)) \ : \ \mathsf{nat}$ is a term (being more or less equivalent to `if(x==0) 0 else y+1` in Scala, if x and y are declared as Ints).

The next rule is that of λ-abstraction. That is, when M is an expression of type τ in a context Γ, in which a (free) variable x of type σ appears (that is, $\Gamma_1, x : \sigma, \Gamma_2 \ \vdash \ M \ : \ \tau$), then one can make this "block" M to be the body of a function taking x as the input parameter. The resulting function is of type $\sigma \to \tau$ (one sets the value of x, which is of type σ, then M evaluates to a value with type τ) and is written as $\lambda x : \sigma.M$. Formally,

$$\frac{\Gamma_1, x : \sigma, \Gamma_2 \ \vdash \ M \ : \ \tau}{\Gamma_1, \Gamma_2 \ \vdash \ \lambda x : \sigma.M \ : \ \sigma \to \tau}$$

For example, $x : \mathsf{nat} \ \vdash \ \lambda y : \mathsf{nat}.\mathsf{ifzero}(x, 0, \mathsf{succ}(y)) \ : \ \mathsf{nat} \to \mathsf{nat}$ is a term (which can be written in Scala as `(y:Int) => if(x==0) 0 else y+1`, if x is declared somewhere out of this scope as an Int).

Having a function M of type $\sigma \to \tau$ and some expression N of type σ in some context, we can apply the function M on the input N, writing $M(N)$, formally:

$$\frac{\Gamma \ \vdash \ M \ : \ \sigma \to \tau, \quad \Gamma \ \vdash \ N \ : \ \sigma}{\Gamma \ \vdash \ M(N) \ : \ \tau}$$

For example, $x : \mathsf{nat} \vdash (\lambda y : \mathsf{nat}.\mathsf{ifzero}(y, 0, \mathsf{succ}(y)))(\mathsf{succ}(x)) : \mathsf{nat}$ is a term, esentially having the same semantics as `{y:Int => if(y==0) 0 y+1}(x+1)` which evaluates to $x+2$ for every natural number x (since $x+1$ is checked to be zero, which is always false, thus the value of this expression becomes $x+1+1$ for each $x \geq 0$).

Finally, we also have a rule for recursion. In λ-calculus this notion is somewhat more complicated than in ordinary programming languages, since here one cannot call a function by its name. Instead, given a definition `def f(n) = M` in which f also appears (that is, a definition of a recursive function), with f being some function of type σ, we can first write the expression $\lambda f : \sigma.M$, in which f becomes a variable. Now for such an expression, the f what we have in mind is a fixed point of this λ-function since if we substitute our intended f, then it satisfies its own (recursively given) specification.

To make this a bit more understandable, suppose we have the expression $M = \mathsf{ifzero}(x, 1, f(x-1)+2)$ in which x is of type nat and f is of type $\mathsf{nat} \to \mathsf{nat}$. That is,

$$f : \mathsf{nat} \to \mathsf{nat}, x : \mathsf{nat} \vdash \mathsf{ifzero}(x, 1, f(x-1)+2) : \mathsf{nat}.$$

What we have in mind: we want to implement the function $f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n-1)+2 & \text{otherwise.} \end{cases}$ One can easily check that this recursive specification is satisfied by the fuction $n \mapsto 2n$. Now we can view this block M as a transformation of $\mathsf{nat} \to \mathsf{nat}$ functions as

$$x : \mathsf{nat} \vdash \lambda f : \mathsf{nat} \to \mathsf{nat}.\mathsf{ifzero}(x, 0, f(x-1)+2) : \mathsf{nat}.$$

That is, a construct which can be implemented in Scala as

```
def transform( f : Int => Int ) : Int => Int = {
  n => if( n == 0 ) 0 else f( n - 1 ) + 2
}
```

So the function $\mathsf{transform}$ above gets some function f and outputs some other function f'. For example, if f is the constant zero function (`n=>0`), then its transformed variant f' returns $f'(0) = 0$ and $f'(n) = 2$ for $n > 0$. Hence, for this function $f \neq f'$, the constant zero is not a fixed point of this transformation. But, the function $n \mapsto 2n$ is: if f is a some arbitrary implementation of $n \mapsto 2n$, then f' outputs $f'(0) = 0$ and $f'(n) = f(n-1)+2 = 2(n-1)+2 = 2n$ for $n > 0$, thus f' is also an implementation of the function $n \mapsto 2n$. Thus, $n \mapsto 2n$ is a fixed point of the $\mathsf{transform}$ function above! Hence, recursive specifications can be modeled this way by fixed points[9]: taking the function of type $\sigma \to \tau$, λ-abstracting its body resulting in some $(\sigma \to \tau) \to (\sigma \to \tau)$ transformation, and taking a fixed point of that transformation. Formally,

$$\frac{\Gamma \vdash M : \sigma \to \sigma}{\Gamma \vdash Y_\sigma(M) : \sigma}$$

and Y_σ is called the recursion operator (of type σ).

Example: $\mathsf{iterate}$ in λ-calculus

For an exercise, let us write our running example function $\mathsf{iterate}$ as a λ-term, building the smallest blockfirst, the built-in 1 in the middle. That's a built-in constant, in a context where are

[9] it will turn out that these are least fixed points, actually

the symbols n (of type Int), f (of type $\mathsf{Int} \to \mathsf{Int}$) and $\mathsf{iterate}$ (of type $(\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int})$) are visible. Thus we write the corresponding term as

$$i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}), f : \mathsf{nat} \to \mathsf{nat}, n : \mathsf{nat} \vdash 1 \ : \ \mathsf{nat}$$

since we have a rule (the second rule in the set for built-ins) that 1 is an expression of type nat in any context. Also, in the same context, f is a $\mathsf{nat} \to \mathsf{nat}$ type variable, thus we can apply the variable evaluation rule as:

$$i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}), f : \mathsf{nat} \to \mathsf{nat}, n : \mathsf{nat} \vdash f \ : \ \mathsf{nat} \to \mathsf{nat}$$

Now since in the same context, f is of type $\mathsf{nat} \to \mathsf{nat}$ and 1 is of type nat, we can use the rule for function application here and get:

$$i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}), f : \mathsf{nat} \to \mathsf{nat}, n : \mathsf{nat} \vdash f(1) \ : \ \mathsf{nat}$$

Similarly, in the very same context, n is an expression of type nat, applying the variable evaluation rule again:

$$i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}), f : \mathsf{nat} \to \mathsf{nat}, n : \mathsf{nat} \vdash n \ : \ \mathsf{nat}$$

Thus, the third built-in rule gives us $\mathsf{pred}n$ is also such an expression:

$$i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}), f : \mathsf{nat} \to \mathsf{nat}, n : \mathsf{nat} \vdash \mathsf{pred}(n) \ : \ \mathsf{nat}$$

Also by variable evaluation we get the type of i:

$$i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}), f : \mathsf{nat} \to \mathsf{nat}, n : \mathsf{nat} \vdash i \ : \ (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat})$$

Since f is still a $\mathsf{nat} \to \mathsf{nat}$ function in this context, we can apply i to f and get:

$$i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}), f : \mathsf{nat} \to \mathsf{nat}, n : \mathsf{nat} \vdash i(f) \ : \ \mathsf{nat} \to \mathsf{nat}$$

As $i(f)$ has type $\mathsf{nat} \to \mathsf{nat}$ and $\mathsf{pred}(n)$ has type nat in this context, we can apply $i(f)$ to $\mathsf{pred}(n)$, getting a nat:

$$i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}), f : \mathsf{nat} \to \mathsf{nat}, n : \mathsf{nat} \vdash (i(f))(\mathsf{pred}(n)) \ : \ \mathsf{nat}$$

As the above expression is of type nat, one can apply f on it:

$$i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}), f : \mathsf{nat} \to \mathsf{nat}, n : \mathsf{nat} \vdash f((i(f))(\mathsf{pred}(n))) \ : \ \mathsf{nat}$$

Then, we have three expressions of type nat in the same context: n, $f(1)$ and $f((i(f))$, we can combine them by an if-else:

$$i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}), f : \mathsf{nat} \to \mathsf{nat}, n : \mathsf{nat} \vdash \mathsf{ifzero}(n, f(1), f((i(f))(\mathsf{pred}(n)))) \ : \ \mathsf{nat}$$

(now we have the "λ-implementation" of the block M inside $i = n \Rightarrow M$). Then, we λ-abstract the argument n (thus we will have the implementation of the function $n \Rightarrow M$):

$$i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}), f : \mathsf{nat} \to \mathsf{nat}$$

$$\vdash$$

$$\lambda n : \mathsf{nat}.(\mathsf{ifzero}(n, f(1), f((i(f))(\mathsf{pred}(n))))) \ : \ \mathsf{nat} \to \mathsf{nat}$$

Then, i is the function that takes f as input and outputs this function $n \Rightarrow M$, thus we λ-abstract the argument f as well:

$$i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat})$$

$$\vdash$$

$$\lambda f : \mathsf{nat} \to \mathsf{nat}.\lambda n : \mathsf{nat}.(\mathsf{ifzero}(n, f(1), f((i(f))(\mathsf{pred}(n)))))$$

$$:$$

$$(\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat})$$

Here comes the "λ-and-Y-trick": we want to construct a recursive function, say, i, to have this expression as body. First we λ-abstract i, yielding the term

$$\vdash$$

$$\lambda i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}).\lambda f : \mathsf{nat} \to \mathsf{nat}.\lambda n : \mathsf{nat}.(\mathsf{ifzero}(n, f(1), f((i(f))(\mathsf{pred}(n)))))$$

$$:$$

$$((\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat})) \to ((\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}))$$

and now we have a function that transforms an input function i into another function i', and the actual $\mathsf{iterate}$ function we want to have is the (least) fixed point of this transformation, so we apply the recursion operator $Y_{(\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat})}$:

$$\vdash$$

$$Y_{(\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat})}(\lambda i : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}).\lambda f : \mathsf{nat} \to \mathsf{nat}.\lambda n : \mathsf{nat}.($$
$$\mathsf{ifzero}(n, f(1), f((i(f))(\mathsf{pred}(n))))))$$

$$:$$

$$(\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat})$$

and that is the λ-term equivalent to our Scala function $\mathsf{iterate}$.

At least, intuitively, since we have not defined the semantics of λ-terms, only the syntax along with some informal intuition.

So the plan for this part is the following:

- We will give an "operational" semantics for λ-terms, that will be a step-wise computational semantics (e.g. if I have an application term $M(N)$ then I should evaluate N first, then substitute the result in place of the formal parameter of M, then evaluate the resulting program, and such) corresponding to our informal intuition above. This gives us "how" should those terms be evaluated, a local viewpoint.
- Then, we will give a "denotational" semantics also, that will assign an actual function (i.e. a mathematical function, a mapping) to each λ-term. This gives us "what" those terms mean, a global viewpoint.
- Finally, we will show that the two semantics are equivalent, thus they complement each other, these are really the "how" and "what" of the same behaviour.

Operational semantics of λ-calculus

Before proceeding with the operational semantics, we first define the so-called substitution of (pure) λ-terms. The semantics is defined on pure terms, that can be generated by the following grammar:

$$M \ \rightarrow \ x \mid \mathsf{0} \mid \mathsf{succ}(M) \mid \mathsf{pred}(M) \mid M(M) \mid \lambda x:\sigma.(M) \mid \mathsf{ifzero}(M,M,M) \mid Y_\sigma(M)$$

where x is some variable and σ is some type. It is easy to check (via structural induction) that whenever $\Gamma \vdash M : \sigma$ is a term, then M is a pure term. At the same time, not all pure terms can be typified: for example, $x(x)$ is a pure term but there is no context Γ and type σ with $\Gamma \vdash x(x) : \sigma$ being a term (since then x should have some type $\tau \rightarrow \sigma$ and the type τ at the same time which is impossible).

Definition: Substitution.

Given pure terms M and N, and a variable x, the substitution $M[x/N]$ is defined via structural induction on M as

- $x[x/N] = N$ for the variable x;
- $y[x/N] = y$ for the variable $y \neq x$;
- $0[x/N] = 0$;
- $(\mathsf{succ}(M'))[x/N] = \mathsf{succ}(M'[x/N])$;
- $(\mathsf{pred}(M'))[x/N] = \mathsf{pred}(M'[x/N])$;
- $(M_1(M_2))[x/N] = (M_1[x/N](M_2[x/N]))$;
- $(\lambda x:\sigma.(M'))[x/N] = \lambda x:\sigma.(M')$;
- $(\lambda y:\sigma.(M'))[x/N] = \lambda y:\sigma.(M'[x/N])$ if $y \neq x$ does not occur in N;
- $(\lambda y:\sigma.(M'))[x/N] = \lambda z:\sigma.(M'[y/z][x/N])$ if $y \neq x$ and y occur in N, where z is a fresh variable;
- $\mathsf{ifzero}(M_1,M_2,M_3)[x/N] = \mathsf{ifzero}(M_1[x/N],M_2[x/N],M_3[x/N])$;
- $(Y_\sigma(M))[x/N] = Y_\sigma(M[x/N])$.

Basically what happens: we get $M[x/N]$ be replacing each free occurrence of x (an occurrence is free if it is not within the scope of some λx) by N, and, if this occurrence of x is within the scope of some λy for some y also occurring in N, then we rename the y in the λy construct to some other, unused variable.

As an example:

Consider the term $M = \mathsf{ifzero}(x, (\lambda x.\mathsf{pred}(x))(\mathsf{succ}(0)), (\lambda y.\mathsf{ifzero}(y,0,x))(0))$ and let us calculate $M[x/\mathsf{succ}(y)]$.

By definition, we get

$$\begin{aligned} & \mathsf{ifzero}\Big(x,\ (\lambda x.\mathsf{pred}(x))(\mathsf{succ}(0)),\ (\lambda y.\mathsf{ifzero}(y,0,x))(0)\Big)[x/\mathsf{succ}(y)] \\ = \ & \mathsf{ifzero}\Big(x[x/\mathsf{succ}(y)],\ (\lambda x.\mathsf{pred}(x))(\mathsf{succ}(0))[x/\mathsf{succ}(y)],\ (\lambda y.\mathsf{ifzero}(y,0,x))(0)[x/\mathsf{succ}(y)]\Big) \end{aligned}$$

and calculating the three arguments one by one,

$$x[x/\mathsf{succ}(y)] = \mathsf{succ}(y),$$

$$\begin{aligned} \big(\ (\lambda x.\mathsf{pred}(x))(\mathsf{succ}(0))\big)[x/\mathsf{succ}(y)] &= \big(\ ((\lambda x.\mathsf{pred}(x))[x/\mathsf{succ}(y)])((\mathsf{succ}(0))[x/\mathsf{succ}(y)])\big) \\ &= \big(\ (\lambda x.\mathsf{pred}(x))(\mathsf{succ}(0[x/\mathsf{succ}(y)]))\big) \\ &= \big(\ (\lambda x.\mathsf{pred}(x))(\mathsf{succ}(0))\big) \end{aligned}$$

and

$$\begin{aligned} (\lambda y.\mathsf{ifzero}(y,0,x))(0)[x/\mathsf{succ}(y)] &= \big((\lambda y.\mathsf{ifzero}(y,0,x))[x/\mathsf{succ}(y)]\big)(0[x/\mathsf{succ}(y)]) \\ &= \big((\lambda z.(\mathsf{ifzero}(y,0,x)[y/z][x/\mathsf{succ}(y)])\big)(0) \\ &= \big((\lambda z.(\mathsf{ifzero}(y[y/z],0[y/z],x[y/z])[x/\mathsf{succ}(y)])\big)(0) \\ &= \big((\lambda z.(\mathsf{ifzero}(z,0,x)[x/\mathsf{succ}(y)])\big)(0) \\ &= \big((\lambda z.(\mathsf{ifzero}(z[x/\mathsf{succ}(y)],0[x/\mathsf{succ}(y)],x[x/\mathsf{succ}(y)]))\big)(0) \\ &= \big((\lambda z.(\mathsf{ifzero}(z,0,\mathsf{succ}(y)))\big)(0) \end{aligned}$$

thus the result is

$$\mathsf{ifzero}\Big(\mathsf{succ}(y),(\lambda x.\mathsf{pred}(x))(\mathsf{succ}(0)),(\lambda z.\mathsf{ifzero}(z,0,\mathsf{succ}(y)))(0)\Big).$$

We can get the very same result as follows: first we highlight the free occurrences of x:

$$\mathsf{ifzero}(x,(\lambda x.\mathsf{pred}(x))(\mathsf{succ}(0)),(\lambda y.\mathsf{ifzero}(y,0,x))(0))$$

then, as $\mathsf{succ}(y)$ contains the variable y, we highlight those λy that contain at least one highlighted x in their scope, along with their own y occurrences:

$$\mathsf{ifzero}(x,(\lambda x.\mathsf{pred}(x))(\mathsf{succ}(0)),(\lambda y.\mathsf{ifzero}(y,0,x))(0))$$

Finally we replace all the x by $N(y)$ and all the y by some new variable:

$$\mathsf{ifzero}(\mathsf{succ}(y),(\lambda x.\mathsf{pred}(x))(\mathsf{succ}(0)),(\lambda z.\mathsf{ifzero}(z,0,\mathsf{succ}(y)))(0)).$$

The result is the same.

Having defined substitution, we are now ready to define the operational semantics of λ-calculus as follows. We'll define a relation $\triangleright$ between pure terms. The relation $M \triangleright N$ can be read as "M gets rewritten to N in one step", represents one computation step. Some terms can be rewritten, some are not. Those that cannot be rewritten are called (syntactic) values.

The rewriting rules are defined as follows:

1. If $M = 0$, $M = x$ for some variable or $M = (\lambda x : \sigma.M')$, then M is already a value and

does not get rewritten.

2. If $M = (\lambda x : \sigma.M')(N)$, i.e., a function application with the function being "in λ-form", then $M \triangleright M'[x/N]$. That is, we substitute the argument N in place of the formal parameter x in the body of the function.

3. If $M = M_1(N)$ and $M_1 \triangleright M_2$, that is, it is a function application but the function can be rewritten (thus it is not in λ-form yet), then we rewrite the function part: $M \triangleright M_2(N)$.

4. If $M = \mathsf{ifzero}(0, M_2, M_3)$, then $M \triangleright M_2$. That is, if the branching condition of M is already evaluated and its value is 0, then we continue on the if branch.

5. If $M = \mathsf{ifzero}(\mathsf{n}+1, M_2, M_3)$, then $M \triangleright M_3$. Here $\mathsf{n}+1$ is a shorthand for the pure term $\mathsf{succ}^{n+1}(0)$. That is, if the branching condition of M is already evaluated and its value is some positive natural number, then we continue on the else branch.

6. If $M = \mathsf{ifzero}(M_1, M_2, M_3)$ and $M_1 \triangleright M_1'$, that is, if the branching condition is not a value yet, then we continue its evaluation and $M \triangleright \mathsf{ifzero}(M_1', M_2, M_3)$.

7. If $M = Y_\sigma(M')$, then $M \triangleright M'(Y_\sigma(M'))$. That is, the recursion operator gets substituted into its body. We'll see an example for that below.

8. If $M = \mathsf{succ}(M_1)$ and $M_1 \triangleright M_2$, then $M \triangleright \mathsf{succ}(M_2)$. That is, if we have a succ-term, we try to rewrite its argument.

9. Similarly, if $M = \mathsf{pred}(M_1)$ and $M_1 \triangleright M_2$, then $M \triangleright \mathsf{pred}(M_2)$.

10. For the numbers, we have $\mathsf{pred}(0) \triangleright 0$. That is, 0 cannot be decremented further, $0 - 1$ becomes 0 again.

11. Also, $\mathsf{pred}(\mathsf{n}+1) \triangleright \mathsf{n}$, that is, the predecessor of $n+1$ is n.

These rules can be written in concise form as follows:

$$\frac{}{(\lambda x : \sigma.M)(N) \ \triangleright \ M[x/N]} \qquad \frac{M \ \triangleright \ M'}{M(N) \ \triangleright \ M'(N)}$$

$$\frac{}{\mathsf{ifzero}(0, M_2, M_3) \ \triangleright \ M_2} \qquad \frac{}{\mathsf{ifzero}(\mathsf{n}+1, M_2, M_3) \ \triangleright \ M_3}$$

$$\frac{M_1 \triangleright M_1'}{\mathsf{ifzero}(M_1, M_2, M_3) \ \triangleright \ \mathsf{ifzero}(M_1', M_2, M_3)}$$

$$\frac{}{Y_\sigma(M) \ \triangleright \ M(Y_\sigma(M))}$$

$$\frac{M \ \triangleright \ N}{\mathsf{succ}(M) \ \triangleright \ \mathsf{succ}(N)} \qquad \frac{M \ \triangleright \ N}{\mathsf{pred}(M) \ \triangleright \ \mathsf{pred}(N)}$$

$$\frac{}{\mathsf{pred}(0) \ \triangleright \ 0} \qquad \frac{}{\mathsf{pred}(\mathsf{n}+1) \ \triangleright \ n}$$

Then, we have defined $\triangleright$. The notion $M \triangleright^* N$ denotes that M can be rewritten into N in 0, 1, or more (finitely many) steps; and $M \Downarrow N$ denotes that $M \triangleright^* N$ and N is a value (i.e., cannot be further reduced). Clearly, since the rules uniquely determine the rewriting process, this N, if exists, is uniquely defined and is called the value of M.

Example: $2 \times 2 = 4$

The doubling function can be defined in Scala as

```
def f( n : Int ) : Int = if( n == 0 ) 0 else 2 + f( n - 1 )
```

which can be written as the term

$$\vdash Y_{\mathsf{nat}\to\mathsf{nat}}\Big(\lambda f : \mathsf{nat}\to\mathsf{nat}.\lambda n : \mathsf{nat}.\mathsf{ifzero}(n, 0, \mathsf{succ}(\mathsf{succ}(f(\mathsf{pred}(n)))))\Big) \ : \ \mathsf{nat}\to\mathsf{nat}.$$

Let us evaluate this function for the input $\mathsf{succ}(\mathsf{succ}(0))$, that is, $\mathsf{2}$. For the sake of readability, we omit type specifications and write ifz for ifzero, s for succ and p for pred: so we want to evaluate the (pure) term

$$\Big(Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))\Big)(\mathsf{2}).$$

Note that we also omit some parentheses around arguments of unary function symbols etc. Then, this term is a function application, with its left-hand side being not in λ-form. Thus we rewrite the part $Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))$ according to the recursion rule and get:

$$\Big((\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n)))\big(Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))\big)\Big)(\mathsf{2}).$$

This term is again an application with the function part being also an application (and not in λ-form yet), thus we rewrite the function part, which is, in turn, also an application, with its function part being in λ-form. So we rewrite the term as follows: we substitute the argument in place of the formal parameter f in the body of the λ-function and get:

$$\Big((\lambda n.\mathsf{ifz}(n, 0, \mathsf{ss}(Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))(\mathsf{p}(n)))))\Big)(\mathsf{2}).$$

Then, our current term is a function application, with the left-hand side being in λ-form. Thus, we substitute the argument $\mathsf{2}$ in place of the free ns in the body of the function and get:

$$\mathsf{ifz}\Big(\mathsf{2}, 0, \mathsf{ss}(Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))(\mathsf{p}(\mathsf{2})))\Big).$$

Observe that according to the substitution rule, we have not replaced the occurrences of n which are in the scope of the inner λn. Then, this term is a conditional branch, with its condition being already evaluated to a nonzero value thus we rewrite it to its else branch and get:

$$\mathsf{ss}(Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))(\mathsf{p}(\mathsf{2}))).$$

We recolored the inner parts or our current term to reflect the fact that it is a succ-term, thus its argument should be rewritten, which happens to be also a succ-term, whose argument is a function application, with the left-hand side not in λ-form. Hence we rewrite the function part according to the recursion rule and get

$$\mathsf{ss}((\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n)))(Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n)))(\mathsf{p}(\mathsf{2}))).$$

In the next step, we have a function application with its left-hand side being in λ-form, thus we apply the substitution:

$$\mathsf{ss}((\lambda n.\mathsf{ifz}(n, 0, \mathsf{ss}(Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))(\mathsf{p}(n)))))(\mathsf{p}(\mathsf{2}))).$$

Then again, inside the two succ operators, we have a function application, with the function being in λ-form. Hence we substitute $\mathsf{p}(2)$ in place of the green n's:

$$\mathsf{ss}(\mathsf{ifz}(\mathsf{p}(2), 0, \mathsf{ss}(Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))(\mathsf{p}(\mathsf{p}(2)))))).$$

Here we have an ifz with a not yet evaluated condition $\mathsf{p}(2)$. We rewrite it to 1:

$$\mathsf{ss}(\mathsf{ifz}(1, 0, \mathsf{ss}(Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))(\mathsf{p}(\mathsf{p}(2)))))).$$

Then, the ifz subterm is rewritten to its else part as $1 \neq 0$:

$$\mathsf{ssss}(Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))(\mathsf{p}(\mathsf{p}(2)))).$$

Then yet again, we have to apply the recursion rule (one last time) and get:

$$\mathsf{ssss}((\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))(Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n)))(\mathsf{p}(\mathsf{p}(2)))).$$

and substituting the argument in place of the formal parameter f we get:

$$\mathsf{ssss}((\lambda n.\mathsf{ifz}(n, 0, \mathsf{ss}((Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))(\mathsf{p}(n))))))(\mathsf{p}(\mathsf{p}(2)))).$$

Then applying a function application fule, we substitute $\mathsf{pp}(2)$ in place of the free ns in the body of the function:

$$\mathsf{ssss}(\mathsf{ifz}(\mathsf{pp}(2), 0, \mathsf{ss}((Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))(\mathsf{p}(\mathsf{pp}(2))))))).$$

The ifzero's condition $\mathsf{pp}(2)$ gets rewritten to $\mathsf{p}(1)$, then to 0:

$$\mathsf{ssss}(\mathsf{ifz}(0, 0, \mathsf{ss}((Y\lambda f.\lambda n.\mathsf{ifz}(n, 0, \mathsf{ssfp}(n))(\mathsf{p}(\mathsf{pp}(2))))))).$$

Which is 0, thus the conditional branch gets (finally) rewritten to its if part:

$$\mathsf{ssss}(0).$$

And that's it. It's the term $\mathsf{4}$. Two times two is four.

Denotational semantics: the interpretation domains

In this part we associate to each term $\Gamma \vdash M : \sigma$ a function denoted $[[\Gamma \vdash M : \sigma]]$. To this end, we first define an interpretation domain $[[\sigma]]$ to each type σ.

It is more or less clear from the previous section that the objects of type natshould be the natural numbers. However, instead of $\mathbb{N}$, we define $[[\mathsf{nat}]]$ to be the poset $\mathbb{N}_\perp$. The intuition is that when a computation does not terminate (such cases naturally occur in every Turing-complete language, e.g. the function `def f(n)=if(n==0) 0 else f(n+1)`) terminates only if it gets 0 as argument, in all the other cases it enters an infinite loop. (As an exercise, the reader is encouraged to construct an equivalent term M of λ-calculus and apply the rewrite rules of the operational semantics to check what happens). Thus, there is no $\mathbb{N} \to \mathbb{N}$ function being "equivalent" to the f above. But if we introduce the element $\perp$ and treat it as the sign of nontermination, then the (mathematical) function $f : \mathbb{N}_\perp \to \mathbb{N}_\perp$ defined as $f(0) = 0$ and $f(n) = \perp$ for each $n \neq \perp$ corresponds exactly to the above implementation.

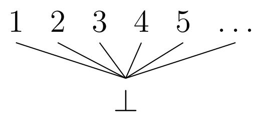
Also, for reasons becoming apparent soon, we want to have not arbitrary functions, but contінous functions appear as values for objects of type $\sigma \to \tau$. We introduce the following notation:

Definition

When P and Q are posets, then $[P \to Q]$ denotes the poset of continuous functions from P to Q.

Let us recall that a function $f : P \to Q$ is continuous if whenever $X \subseteq P$ is a nonempty linearly ordered subset of P having a supremum $\bigvee X$, then $f(\bigvee X) = \bigvee_{x \in X} f(x)$. Of course these functions form a poset with respect to the pointwise ordering $f \leq g \Leftrightarrow \forall n \in P\ f(n) \leq g(n)$.

Also recall that $\mathbb{N}_\perp$ is the poset where $\perp$ is the least element (that is, $\perp \leq n$ for each $n \in \mathbb{N}$) and the integers are pairwise incomparable which can be depicted as



Clearly, $\mathbb{N}_\perp$ is not "that nice" in the sense it is not a complete lattice: e.g. the subset $\{1, 2\}$ has no upper bound (thus has no supremum). However, it is a so-called complete poset:

Definition: Complete poset, CPO.

A poset P is called a complete poset, or CPO, if every linearly ordered subset of P has a supremum.

That is, if $X \subseteq P$ is a subset of P such that for every pair $x, y \in X$ of member of X we either have $x \leq y$ or $y \leq x$, then $\bigvee X$ exists. In particular, since $\emptyset$ is a linearly ordered subset, $\bigvee \emptyset$ also exists in a CPO, that is, any CPO has a least element.

Observe that if $f : P \to Q$ is a function where P is a CPO, then a nonempty, linearly ordered subset X of P always have a supremum. Thus, this last condition can be removed when we test continuity.

A nice property of continuous functions between CPOs is that they form a CPO as well:

Proposition

If P and Q are CPOs, then so is $[P \to Q]$.

Proof

Let $F \subseteq [P \to Q]$ be a nonempty, linearly ordered subset of $[P \to Q]$, that is, $F = \{f_i : i \in I\}$ with each f_i being a continuous function from P to Q. We show that $\bigvee F$ exists and is continuous.

For existence, we already know that $\bigvee F : P \to Q$ can be defined as $(\bigvee F)(x) = \bigvee_{i \in I}(f_i(x))$, assuming the supremum on the right-hand side exists. Since $F(x) = \{f_i(x) : i \in I\}$ is a subset of the CPO Q, it suffices to check that this set is linearly ordered. So let $f_i(x), f_j(x) \in F(x)$ be two elements of $F(x)$. Since F is linearly ordered, we either have $f_i \leq f_j$ or $f_j \leq f_i$; by definition this implies either $f_i(x) \leq f_j(x)$ or $f_j(x) \leq f_i(x)$, as desired. Thus, $\bigvee F$ exists.

(Observe that up till this point we used only that Q is a CPO. Hence, for any poset P and CPO Q, the set of functions $P \to Q$ is a CPO.)

To show that $\bigvee F$ is continuous, let $X \subseteq P$ be a nonempty linearly ordered subset of P. Since P is a CPO, it has a supremum $\bigvee X$. We have to show that $(\bigvee F)(\bigvee X) = \bigvee_{x\in X}(\bigvee F)(x)$. Which holds, since

$$\begin{aligned}
(\bigvee F)(\bigvee X) &= (\bigvee_{i\in I} f_i)(\bigvee_{x\in X} x) && \text{spelling out details}\\
&= \bigvee_{i\in I}(f_i(\bigvee_{x\in X} x)) && \text{definition of } \bigvee_{i\in I} f_i\\
&= \bigvee_{i\in I}\bigvee_{x\in X} f_i(x) && \text{applying continuity of } f_i\\
&= \bigvee_{x\in X}\bigvee_{i\in I} f_i(x) && \text{taking suprema can be swapped}^\star\\
&= \bigvee_{x\in X}(\bigvee_{i\in I} f_i)(x) && \text{definition of } \bigvee_{i\in I} f_i\\
&= \bigvee_{x\in X}(\bigvee F)(x).
\end{aligned}$$

Hence, $\bigvee F$ belongs to $[P \to Q]$ as well, thus it is a CPO.

Note that when we swapped the suprema operators, we implicticly assumed that the suprema $\bigvee_{i\in I} f_i(x)$ exist – but that holds, since Q is a CPO and F is linearly ordered.

Thus, if we define our interpretation domains as follows:

Definition: Interpretation domains of types.

For a type σ we define its interpretation domain $[[\sigma]]$ recursively as follows:

- $[[\mathsf{nat}]] = \mathbb{N}_\perp$
- for a functional type $\sigma \to \tau$, let $[[\sigma \to \tau]]$ be the poset $\Big[\ [[\sigma]] \to [[\tau]]\ \Big]$ of functions.

That is, an object of type $\sigma \to \tau$ should be a continuous function taking an argument of type $[[\sigma]]$ and mapping it into $[[\tau]]$.

Then by the previous proposition we have:

Proposition

For any type σ, the poset $[[\sigma]]$ is a CPO.

Proof

To see that $[[\mathsf{nat}]] = \mathbb{N}_\perp$ is a CPO, let us enumerate the linearly ordered subsets. Since such a set cannot contain two different natural numbers (as these are pairwise incomparable), the only such sets are $\emptyset$ and the singleton sets $\{x\}$ (these are linearly ordered subsets of any poset), and the sets of the form $\{\perp, n\}$ with $n \in \mathbb{N}$. For the sets $\{x\}$ we have $\bigvee\{x\} = x$, for $\emptyset$ we have $\bigvee\emptyset = \perp$, which is also fine, and for the sets $\{\perp, n\}$ we have $\bigvee\{\perp, n\} = n$. Thus $\mathbb{N}_\perp$ (and in fact, any poset of the form $X_\perp$) is a CPO.

For the induction case we just apply our previous proposition: if $[[\sigma]]$ and $[[\tau]]$ are CPOs, then so is $[[\sigma \to \tau]] = \Big[[[\sigma]] \to [[\tau]]\Big]$.

Also, to a context $\Gamma = x_1 : \sigma_1, x_2 : \sigma_2, \dots, x_n : \sigma_n$ we also associate an interpretation domain $[[\Gamma]]$. Since a context essentially declares the variables $x_1, \dots, x_n$, having respectively the types $\sigma_1, \dots, \sigma_n$, it makes sense to say that Γ should be interpreted by an assignment: each x_i should get a value from the corresponding interpretation domain $[[\sigma]]$. Thus, the correct definition is

Definition: Interpretation domains of contexts.

For a context $\Gamma = x_1 : \sigma_1, \dots, x_n : \sigma_n$, let us define its interpretation domain as $[[\Gamma]] = [[\sigma_1]] \times [[\sigma_2]] \times \dots \times [[\sigma_n]]$.

That is, members of $[[\Gamma]]$ are n-tuples of objects, at coordinate i there is a member of the corresponding domain $[[\sigma_i]]$. Of course $[[\Gamma]]$ is also a poset, equipped with the pointwise ordering and moreover, is a CPO as well, since the following holds:

Proposition

If each P_i is a CPO for $i \in I$, then so is $P = \prod_{i\in I} P_i$.

Proof

Let $U \subseteq P$ be a linearly ordered subset of the product. We have to prove that $\bigvee U$ exists. Now for each $i \in I$, the supremum of the set $U(i) = \{u(i) : u \in U\}$ (that is, the supremum of the values at the ith coordinates) exists, since for any $u(i), v(i) \in U(i)$ we either have $u \leq v$ or $v \leq u$ since U is linearly ordered, implying either $u(i) \leq v(i)$ or $v(i) \leq u(i)$. Thus $U(i) \subseteq P_i$ is linearly ordered and since P_i is a CPO, its supremum exists.

But then $(\bigvee U)(i) = \bigvee(U(i))$ is well-defined, thus $\bigvee U$ indeed exists.

Examples: continuous functions

Before proceeding, it is worth to see some continuous functions. We already know the members of $[[\mathsf{nat}]] = \mathbb{N}_\perp$. What functions are the members of $[[\mathsf{nat} \to \mathsf{nat}]]$? This poset is defined as $[[[\mathsf{nat}]] \to [[\mathsf{nat}]]]$, so if $f \in [[\mathsf{nat} \to \mathsf{nat}]]$, then f is a continuous $\mathbb{N}_\perp \to \mathbb{N}_\perp$ function. Several functions from $\mathbb{N}_\perp$ to $\mathbb{N}_\perp$ are given below:

	$\perp$	0	1	2	3	4	5	...
f	$\perp$	4	$\perp$	5	3	4	$\perp$	...
g	3	3	3	3	3	3	3	...
h	0	1	1	1	2	2	$\perp$	...
i	$\perp$	4	2	5	3	4	$\perp$	...

The question is, which of the above are continuous and which are not.

According to the definition of continuity, f is continuous if whenever X is a nonempty, linearly ordered subset of $\mathbb{N}_\perp$, then $f(\bigvee X)$ should coincide with $\bigvee_{x\in X} f(x)$. We already know that this condition is satisfied by the singleton sets, so we only have to check the sets X of the form $\{\perp, n\}$ for some $n \in \mathbb{N}$. Then $\bigvee X$ is n. Hence, the condition for a $\mathsf{nat} \to \sigma$ function f is

$$f(n) = f(n) \vee f(\perp),$$

which is further equivalent to

$$f(\perp) \leq f(n)$$

for each $n \in \mathbb{N}$. This condition is easy to check for the above example functions: if $f(\perp) = \perp$, then $f(\perp) \leq f(n)$ for each n, thus f and i in the example are continuous; if $f(\perp) = m$ for some $m \in \mathbb{N}$, then $m \leq f(n)$ if and only if $f(n) = m$ for each $n \in \mathbb{N}$ (as each $m \in \mathbb{N}$ is a maximal element of $\mathbb{N}_\perp$). Thus, if $f(\perp) = m$ for some $m \in \ltimes$, then f is continuous if and only if f is a constant mapping. Hence, h is not continuous and g (which is presumably the constant 3 function) is continuous. Clearly, constant functions are always continuous.

So f, g and i above are members of $[[\mathsf{nat} \to \mathsf{nat}]]$. This is also a poset and is equipped with the pointwise ordering, according to which $f \leq i$ holds (at least for this part of the table) since $f(x) \leq i(x)$ for each $x \in \mathbb{N}_\perp$. In general, if $f \leq g$ for the functions f, g, then if $f(x) = n$ for some $n \in \mathbb{N}$, then $f(x) = g(x)$ since n is a maximal element of $\mathbb{N}_\perp$. Hence, if f and g are members of some domain $[[\sigma \to \mathsf{nat}]]$ with $f \leq g$, then g is "more specified" than f, i.e., whenever $f(x) \neq \perp$, then $f(x) = g(x)$.

As a last example, let $F : [[\mathsf{nat} \to \mathsf{nat}]] \to [[\mathsf{nat}]]$ be the function $F(f) = \min_{n \in \mathbb{N}}\{f(n) \neq \perp\}$ that is, which returns the least n for which $f(n)$ is an integer. If there is no such n (i.e., f is the constant $\perp$ function), then $F(f)$ is $\perp$.

We claim that this particular function F is not continuous. Indeed, let f be the function in our example above and let j be the same as f with the modification $f(0) = \perp$. Then $j \leq f$ and both functions are continuous. Then, $X = \{j, f\}$ is a linearly ordered subset of $[[\mathsf{nat} \to \mathsf{nat}]]$, with the supremum f. If F were continuous, then it should satisfy $F(f) = F(f) \vee F(j)$ but as $F(f)$ is 0 and $F(j)$ is 2, the supremum on the right hand side does not exists (in $\mathbb{N}_\perp$, the set $\{0, 2\}$ has no upper bound at all). Thus F is not a continuous function.

It will turn out in the following sections that λ-terms can denote continuous functions only. Since λ-calculus is a so-called "Turing-complete" formalism, meaning informally that any program, written in any "real-life" programming language, can be transformed into a λ-term. Hence, the above reasoning can be seen as an undecidability result: it shows that there is no program (in any programming language) that takes some other program f as input, and outputs the least value n for which f does not enter an infinite loop. Essentially, this can be seen as a proof of the so called Halting Problem.

Denotational semantics: semantics of terms

In this section we will associate to each term $\Gamma \vdash M : \sigma$ a continuous function $[[\Gamma \vdash M : \sigma]] \in [[[[\Gamma]] \to [[\sigma]]]]$. The intuition is that such a term evaluates to some element of type σ (that's where the $\to [[\sigma]]$ part comes from), after we supply values to each freely occurring variable of M (that's where the $[[\Gamma]] \to$ part comes from since Γ contains all the freely occurring variables of M).

Our aim is to define this semantics to "coincide" with the operational semantics. That is,

$$\text{Whenever } M \Downarrow V, \text{ then } [[\Gamma \vdash M : \sigma]] = [[\Gamma \vdash V : \sigma]].$$

In this section, let Γ stand for the context $x_1 : \sigma_1, \dots, x_n : \sigma_n$ and $d_i \in [[\sigma_i]]$ for each $1 \le i \le n$.

We start with the case of variable evaluations as: let $[[\Gamma \vdash x_i : \sigma_i]]$ be the ith projection, that is:

$$[[\Gamma \vdash x_i : \sigma_i]](d_1, \dots, d_n) = d_i.$$

The definition is okay, since it is a $[[\Gamma]] \to [[\sigma_i]]$ function and we also know that projections are always continuous.

The next case is that of constant 0 which is also straightforward:

$$[[\Gamma \vdash 0 : \mathsf{nat}]](d_1, \dots, d_n) = 0.$$

Again, it is clearly okay, since it is a $[[\Gamma]] \to [[\mathsf{nat}]]$ function and we also know that constant functions are always continuous.

For successor and predecessor, we define the functions $[[\mathsf{pred}]]$ and $[[\mathsf{succ}]]$ as

$$[[\mathsf{succ}]](x) = \begin{cases} \bot & \text{if } x = \bot \\ x+1 & \text{otherwise,} \end{cases} \qquad [[\mathsf{pred}]](x) = \begin{cases} \bot & \text{if } x = \bot \\ 0 & \text{if } x = 0 \\ x-1 & \text{otherwise.} \end{cases}$$

Since these are $[[\mathsf{nat} \to \mathsf{nat}]]$ functions and the image of $\bot$ is $\bot$ in both cases, they are continuous functions.

Now it makes sense to define the pred and succ terms' semantics as:

$$[[\Gamma \vdash \mathsf{pred}(M) : \mathsf{nat}]](d_1, \dots, d_n) = [[\mathsf{pred}]]\Big([[\Gamma \vdash M : \mathsf{nat}]](d_1, \dots, d_n)\Big)$$
$$[[\Gamma \vdash \mathsf{succ}(M) : \mathsf{nat}]](d_1, \dots, d_n) = [[\mathsf{succ}]]\Big([[\Gamma \vdash M : \mathsf{nat}]](d_1, \dots, d_n)\Big)$$

This definition is also okay: if $\mathsf{pred}(M)$ ($\mathsf{succ}(M)$, respectively) is a term in a context, then so is M, and both are of type nat. By induction on the structure of the term, $[[\Gamma \vdash M : \mathsf{nat}]]$ is a continuous $[[\Gamma]] \to [[\mathsf{nat}]]$ function, while pred and succ are continuous $[[\mathsf{nat}]] \to [[\mathsf{nat}]]$ functions, hence their composition is also a continous $[[\Gamma]] \to [[\mathsf{nat}]]$ function.

At this point it might worth observing that for any n,

$$[[\Gamma \vdash \mathsf{succ}^n(0) : \mathsf{nat}]]$$

is the constant n function which is good since that's the value we want for those terms.

For the conditional branching, we define the function $[[\mathsf{ifzero}]] : \mathbb{N}_\bot^3 \to \mathbb{N}_\bot$ as

$$[[\mathsf{ifzero}]](x, y, z) = \begin{cases} \bot & \text{if } x = \bot, \\ y & \text{if } x = 0, \\ z & \text{otherwise.} \end{cases}$$

The $\bot$ part might seem a bit arbitrary but it has a reason: if we think of $\bot$ as the sign of nontermination, then the first case says that if the calculation of the condition does not terminate, then the whole branching's computation is nonterminating as well, which is fine.

Now we want to show that the $[[\mathsf{ifzero}]]$ function above is continuous. However, it is a $\mathbb{N}_\bot^3 \to \mathbb{N}_\bot$ function, so there are lots of cases (compared to the $\mathbb{N}_\bot \to \mathbb{N}_\bot$ case), e.g. one has to check all the sets X of the form $X = \{(\bot, \bot, \bot), (\bot, y, \bot), (x, y, \bot), (x, y, z)\}$ as these are also linearly ordered (so the problem is that while in $\mathbb{N}_\perp$ there were only the sets of the form $\{\perp, n\}$ which were easy to check, in this poset $\mathbb{N}_\perp^3$ the linearly ordered subsets can be larger. (Actually, the largest such sets can have size 4 but anyways, four is a much larger number than two when it comes to case analysis.)

To circumvent a lengthy case analysis we will show that it suffices to check continuity for only one coordinate at a time. Formally:

Definition

We say that a function $f : \prod_{i\in I} P_i \to Q$ is continuous at coordinate $i \in I$ if whenever $U \subseteq \prod_{i\in I} P_i$ is a linearly ordered subset such that for each $u, v \in U$ and $j \in I$ with $j \neq i$ we have $u(j) = v(j)$, and X has a supremum, then $f(\bigvee X) = \bigvee_{x\in X} f(x)$.

In other words, if we fix the values at each coordinate $j \neq i$ and let the ith coordinate values' range over some linearly ordered set $X_i \subseteq P_i$, then the image of the supremum is the same as the supremum of the images. By this definition it is clear that if f is continuous, then it is continuous at all of its coordinates.

The reverse direction does not hold in general. To see this, let us consider the following function $f : \mathbf{2}^{\mathbb{N}} \to \mathbf{2}$: let $f(x_1, x_2, x_3, \ldots)$ be 0 if an infinite number of x_i's are 1, and 1 otherwise.

Then f is continuous on all of its coordinates: if we fix $x_1, x_2, \ldots, x_{i-1}, x_{i+1}, x_{i+2}, \ldots$ to some values $x_j \in \{0, 1\}$, then whatever x_i is, it is either the case that the sequence of the fixed values already contains an infinite number of 1s (in which case f takes the value 0), or only a finite number of them (in which case even with x_i, there can be only a finite number of them as well in the whole sequence and f takes the value 1), thus $f(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots)$ is a constant function which is continuous.

On the other side, consider the sequence $u_1, u_2, u_3, \ldots$ defined as follows: the first i entries of u_i are 1's, the rest are zeros. Then we have $u_1 \leq u_2 \leq u_3 \leq \ldots$, thus $U = \{u_i : 1 \leq i\}$ is a linearly ordered set and it has the supremum $(1, 1, 1, 1, \ldots)$ since eventually all the coordinates become 1. However, each u_i contains only a finite number of 1s, thus $f(u_i) = 1$ for each i. Thus, $f(\bigvee U) = 0$ and $\bigvee_{u\in U} f(u) = 1$, and f is not continuous.

Note also that f is not even monotone in this setting but it is monotone at each coordinate.

But, if there is only a finite number of coordinates, then the converse also holds:

Proposition

Let $P = \prod_{i=1}^{n} P_i$ be a finite product of posets and $f : P \to Q$ a function.

Then f is continuous (monotone, resp.) if and only if it is continuous (monotone, resp.) at each coordinate $i = 1, \ldots, n$.

Proof

First we show the claim for monotonicity. Assume $f : P \to Q$ is monotone at each coordinate $i = 1, \ldots, n$ and let $(x_1, \ldots, x_n) \leq (y_1, \ldots, y_n)$, that is, $x_i \leq y_i$ for each i.

Then, applying coordinate-wise monotonicity, one at a time, we get

$$\begin{aligned} f(x_1,\ldots,x_n) &\leq f(y_1,x_2,x_3,\ldots,x_n) \\ &\leq f(y_2,y_2,x_3,\ldots,x_n) \\ &\leq \ldots \\ &\leq f(y_1,y_2,y_3,\ldots,y_n), \end{aligned}$$

so f is indeed monotone.

Now assume f is continuous at each coordinate $i=1,\ldots,n$. (Then, f is monotone at each coordinate, thus f is monotone.)

Let $U\subseteq P$ be a nonempty, linearly ordered subset of P having the supremum $u^*=\bigvee U$.

Let us write $U=\{(x_1^i,x_2^i,\ldots,x_n^i):i\in I\}$. Then, $u^*=\left(\bigvee_{i\in I}x_1^i,\bigvee_{i\in I}x_2^i,\ldots,\bigvee_{i\in I}x_n^i\right)$.

Then, applying continuity at one coordinate each, we get

$$\begin{aligned} f(u^*) &= f\Big(\bigvee_{i\in I}x_1^i,\bigvee_{i\in I}x_2^i,\ldots,\bigvee_{i\in I}x_n^i\Big) \\ &= \bigvee_{i_1\in I} f\Big(x_1^{i_1},\bigvee_{i\in I}x_2^i,\ldots,\bigvee_{i\in I}x_n^i\Big) \\ &= \bigvee_{i_1\in I}\bigvee_{i_2\in I} f\Big(x_1^{i_1},x_2^{i_2},\ldots,\bigvee_{i\in I}x_n^i\Big) \\ &= \ldots \\ &= \bigvee_{i_1\in I}\bigvee_{i_2\in I}\ldots\bigvee_{i_n\in I} f(x_1^{i_1},x_2^{i_2},\ldots,x_n^{i_n}). \end{aligned}$$

Then, since each member $u=f(x_1^i,\ldots,x_n^i)$ of U is of the form $f(x_1^{i_1},x_2^{i_2},\ldots,x_n^{i_n})$, we get that $f(u)\leq f(u^*)$ for each $u\in U$, thus $\bigvee_{u\in U}f(u)\leq f(u^*)$.

Also, let $i_1,\ldots,i_n\in I$ and consider the subset $S=\{(x_1^{i_j},\ldots,x_n^{i_j}):1\leq j\leq n\}$ of U. Since U is linearly ordered, so is its finite subset S, which has a largest element $(x_1^{i_j},\ldots,x_n^{i_j})$. But then each vector $(x_1^{i_1},x_2^{i_2},\ldots,x_n^{i_n})$ has some upper bound of the form $(x_1^{i_j},\ldots,x_n^{i_j})$, that is, some upper bound in U. Hence, since f is monotone, we get that each member of the set $\{f(x_1^{i_1},x_2^{i_2},\ldots,x_n^{i_n}):i_1,\ldots,i_n\in I\}$ has some upper bound in $\{f(u):u\in U\}$, thus $f(u^*)\leq\bigvee_{u\in U}f(u)$ as well, hence $f(u^*)=\bigvee_{u\in U}f(u)$ indeed holds.

Now we are ready to proceed with $[[\mathsf{ifzero}]]$:

Proposition

The function $[[\mathsf{ifzero}]]$ is continuous.

Proof

By the previous proposition it suffices to show that $[[\mathsf{ifzero}]]$ is continuous at each of its three coordinates. Now if we fix two of the three coordinates, then we get a function f from $\mathbb{N}_\perp$ to $\mathbb{N}_\perp$ and we already know that such functions are continuous if and only if $f(\perp)\leq f(n)$ for each $n\in\mathbb{N}$. Thus we'll check this property in all the three cases.

1. If we fix y and z, we have to check $[[\mathsf{ifzero}]](\perp,y,z)\leq[[\mathsf{ifzero}]](x,y,z)$ which holds,

since $[[\mathsf{ifzero}]](\bot, y, z) = \bot$ is the least element of $\mathbb{N}_\bot$.

2. If we fix x and z, we have to check $[[\mathsf{ifzero}]](x, \bot, z) \leq [[\mathsf{ifzero}]](x, y, z)$. If $x \neq 0$, then the two values coincide. If $x = 0$, then we get $\bot \leq y$ which clearly holds.

3. Finally, fixing x and y, we check $[[\mathsf{ifzero}]](x, y, \bot) \leq [[\mathsf{ifzero}]](x, y, z)$, which holds since for $x = \bot$ and $x = 0$ the two values coincide (being $\bot$ and y respectively in the two cases), and when x is some other integer, then we get $\bot \leq z$ which also holds.

Hence, $[[\mathsf{ifzero}]]$ is continuous at each of its three coordinates, thus is continuous.

Thus if we define the semantics for conditional branching as

$$[[\Gamma \vdash \mathsf{ifzero}(M_1, M_2, M_3) : \sigma]](d_1, \ldots, d_n) = [[\mathsf{ifzero}]](v_1, v_2, v_3)$$

where $v_i = [[\Gamma \vdash M_i : \mathsf{nat}]](d_1, \ldots, d_n)$ for $i = 1, 2, 3$, then the resulting function is also continuous (applying the induction hypothesis on M continuity of $[[\mathsf{ifzero}]]$ and that target tupling of three functions and function composition preservers continuity).

For defining semantics for function application, we define the following function

$$\mathsf{eval} : \Big([P \to Q] \times P\Big) \to Q : \ \mathsf{eval}(f, x) = f(x)$$

for the CPOs P and Q.

As usual, first we show that this function is continuous:

Proposition

The above evaluation function eval is continuous.

Proof

As eval is a binary function, it suffices to show that it's continuous at both coordinates.

1. Let us fix $x \in Q$ and let $F \subseteq [P \to Q]$ be a nonempty linearly ordered set of continuous functions from P to Q. We have to show that $\mathsf{eval}(\bigvee F, x) = \bigvee_{f \in F} \mathsf{eval}(f, x)$. But as the left hand side is $(\bigvee F)(x)$ (by definition of eval) which further equals to $\bigvee_{f \in F} f(x)$ (by definition of supremum of functions), which is again $\bigvee_{f \in F} \mathsf{eval}(f, x)$ (by definition of eval), this case is verified.

2. Now let us fix $f \in [P \to Q]$ and let $X \subseteq Q$ be a nonempty linearly ordered subset of Q. Then, $\mathsf{eval}(f, \bigvee X) = f(\bigvee X)$ (by definition of eval), which further equals to $\bigvee_{x \in X} f(x)$ (as f is continuous) which is, by definition of eval, $\bigvee_{x \in X} \mathsf{eval}(f, x)$. Thus this case is also verified and eval is continuous.

Hence, if we define the semantics for function application as

$$[[\Gamma \vdash M(N) : \sigma]](d_1, \ldots, d_n) = \mathsf{eval}\Big([[\Gamma \vdash M : \tau \to \sigma]](d_1, \ldots, d_n), [[\Gamma \vdash N : \tau]](d_1, \ldots, d_n)\Big)$$

we also get that (applying induction on the structure of the term, the fact that eval is continuous, and that target tupling of two functions and function composition preserves continuity) this function is also continuous.

In the next step, we define semantics for λ-abstraction. To do that, we first define the curry function (which we have already used in our introductory Scala snippet) as follows: when f is some $(P_1 \times \ldots \times P_n) \to Q$ function for some $n \geq 1$, then let $\mathsf{curry}(f)$ be the $(P_1 \times \ldots \times P_{n-1}) \to (P_n \to Q)$ function defined as

$$\big(\mathsf{curry}(f)(p_1, \ldots, p_{n-1})\big)(p_n) = f(p_1, \ldots, p_n).$$

Again, we show that first if f is continuous, then so is the resulting function $\mathsf{curry}(f)$:

Proposition

The curry function is a $[[(P_1 \times \ldots \times P_n) \to Q] \to [(P_1 \times \ldots \times P_{n-1}) \to [P_n \to Q]]]$ function.

Proof

There are actually three sub-statements to verify in the above sentence:

1. For each $f \in [(P_1 \times \ldots \times P_n) \to Q]$ and values $p_1 \in P_1$, $p_2 \in P_2$, $\ldots$, $p_{n-1} \in P_{n-1}$, the function $\mathsf{curry}(f)(p_1, \ldots, p_n) : P_n \to Q$ is continuous. (That's the innermost bracket.)

   But that's clear since $\mathsf{curry}(f)(p_1, \ldots, p_{n-1})$ is simply the function we get by fixing each coordinate $1 \leq i \leq n-1$ of f to the constant p_i, that is, continuity of this function is exactly the continuity of f at the nth coordinate which holds since f is continuous.

2. For each $f \in [(P_1 \times \ldots \times P_n) \to Q]$, the function $\mathsf{curry}(f)$ is continuous. (That's the middle bracket.)

   To show this, it suffices to show that $\mathsf{curry}(f)$ is continuous at each of its coordinates $i = 1, \ldots, n-1$, that is, whenever $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_{n-1}$ are fixed, and $X \subseteq P_i$ is some nonempty linearly ordered set, then

   $$\mathsf{curry}(f)(p_1, \ldots, p_{i-1}, \bigvee X, p_{i+1}, \ldots, p_{n-1}) = \bigvee_{x \in X} \mathsf{curry}(f)(p_1, \ldots, p_{i-1}, x, p_{i+1}, \ldots, p_{n-1}).$$

   Now two functions are the same if and only if they agree on every input p_n, thus this is further equivalent to

   $$\mathsf{curry}(f)(p_1, \ldots, p_{i-1}, \bigvee X, p_{i+1}, \ldots, p_{n-1})(p_n) = \bigvee_{x \in X} \mathsf{curry}(f)(p_1, \ldots, p_{i-1}, x, p_{i+1}, \ldots, p_{n-1})(p_n),$$

   which is by the definition of curry further equivalent to

   $$f(p_1, \ldots, p_{i-1}, \bigvee X, p_{i+1}, \ldots, p_{n-1}, p_n) = \bigvee_{x \in X} f(p_1, \ldots, p_{i-1}, x, p_{i+1}, \ldots, p_{n-1}, p_n),$$

   which holds since f is continuous, thus it is continuous at its ith coordinate as well.

3. The curry function itself is continuous. (That's the outermost bracket.)

   Let $F \subseteq [(P_1 \times \ldots \times P_n) \to Q]$ be a linearly ordered set of continuous functions. We have to show that $\mathsf{curry}(\bigvee F) = \bigvee_{f \in F} \mathsf{curry}(f)$. By definition, these ($n-1$-ary) functions coincide iff they agree on every possible input vector $p_1, \ldots, p_{n-1}$. As their results are $P_n \to Q$ functions, "agreeing on it" means that these functions have to agree on every possible input p_n. Thus, $\mathsf{curry}(\bigvee F) = \bigvee_{f \in F} \mathsf{curry}(f)$ holds if and only if

$$\mathsf{curry}(\bigvee F)(p_1, \ldots, p_{n-1})(p_n) = \Big(\bigvee_{f \in F} \mathsf{curry}(f)\Big)(p_1, \ldots, p_{n-1})(p_n).$$

Which holds since

$$\begin{aligned}
& \mathsf{curry}(\bigvee F)(p_1, \ldots, p_{n-1})(p_n) \\
= \; & (\bigvee F)(p_1, \ldots, p_n) \quad \text{by definition of } \mathsf{curry} \\
= \; & \bigvee_{f \in F} f(p_1, \ldots, p_n) \quad \text{by definition of function supremum} \\
= \; & \bigvee_{f \in F} \mathsf{curry}(f)(p_1, \ldots, p_{n-1})(p_n) \quad \text{by definition of } \mathsf{curry} \\
= \; & \Big(\bigvee_{f \in F} \mathsf{curry}(f)(p_1, \ldots, p_{n-1})\Big)(p_n) \quad \text{by definition of function supremum} \\
= \; & \Big(\bigvee_{f \in F} \mathsf{curry}(f)\Big)(p_1, \ldots, p_{n-1})(p_n) \quad \text{by definition of function supremum.}
\end{aligned}$$

Hence, if we define the semantics for λ-abstraction as

$$[[\Gamma \vdash \lambda x : \sigma.(M) \;:\; \sigma \to \tau]](d_1, \ldots, d_n) = \mathsf{curry}([[\Gamma, x : \sigma \vdash M : \tau]])(d_1, \ldots, d_n)$$

we get that the resulting function is also continuous (applying the induction hypothesis, the continuity of curry and that function composition preserves continuity).

(A minor note: here we assume that x does not occur in Γ. If it does, then first one should rename the λ-bound variable to some fresh name and apply currying afterwards.)

In that case, the function types work out smoothly: then $[[\Gamma, x : \sigma \vdash M : \tau]]$ is a function from $[[\sigma_1]] \times \ldots [[\sigma_n]] \times [[\sigma]]$ to $[[\tau]]$, thus its curry is a function from $[[\sigma_1]] \times \ldots [[\sigma_n]] = [[\Gamma]]$ to $[[[\sigma]] \to [[\tau]]] = [[\sigma \to \tau]]$ which is exactly what we expect.

Finally, to deal with recursion, we introduce the lfp (for least fixed point) operator, which takes as argument a function $f : P \to [Q \to Q]$ for the CPOs P and Q, and returns a function $\mathsf{lfp}(f) : P \to Q$ defined as follows: let $\big(\mathsf{lfp}(f)\big)(p)$ be the least fixed point of the function $f(p)$.

The definition makes sense since for each $p \in P$, as $g = f(p)$ is a $Q \to Q$ continuous function and has a least fixed point (namely, $\bigvee_{n<\omega} g^n(\bot)$ where $\bot$ is the least element of Q due to Tarski's Fixed Point Theorem). Thus, we can set $\mathsf{lfp}(f)(p)$ to this least fixed point.

Again, we show that this operator lfp preserves continuity:

Proposition

If $f \in [P \to [Q \to Q]]$, then $\mathsf{lfp}(f) \in [P \to Q]$.

Proof

Let $X \subseteq P$ be a linearly ordered nonempty subset of P and let x^* stand for the supremum $\bigvee X$. We have to show that $\mathsf{lfp}(f)(x^*) = \bigvee_{x \in X} \mathsf{lfp}(f)(x)$.

Recall that if $g \in [Q \to Q]$, then its least fixed point is $\bigvee_{n<\omega} g^n(\bot)$. Now if we calculate $\mathsf{lfp}(f)(x^*)$, then this g is the function $f(x^*) = f(\bigvee X) = \bigvee_{x\in X} f(x)$ since f is continuous. That is, the supremum of the functions $f(x)$. (Note that since f is continuous and X is linearly ordered, so is $f(x)$ and as $[Q \to Q]$ is a CPO, we have that this function exists.)

That is, $g(\bot)$ is $\bigvee_{x_1\in X}(f(x_1)(\bot))$.

Calculating $g^2(\bot)$ we get then $g^2(\bot) = \bigvee_{x_2\in X}\bigvee_{x_1\in X} f(x_2)\big(f(x_1)(\bot)\big)$ and in general,

$$g^n(\bot) = \bigvee_{x_n\in X}\bigvee_{x_{n-1}\in X}\dots\bigvee_{x_1\in X} f(x_n)f(x_{n-1})\dots f(x_1)(\bot),$$

that is,

$$g^n(\bot) = \bigvee_{x_1,\dots,x_n\in X} f(x_n)f(x_{n-1})\dots f(x_1)(\bot).$$

Now recall that X is a linearly ordered subset of P and f is continuous, thus it is monotone. Then, whenever $x_1,\dots,x_n \in X$ is a finite subset, then there is a largest element x_i among them. For this x_i we have $f(x_j) \leq f(x_i)$ for each $j = 1,\dots,n$, that is, $f(x_j)(y) \leq f(x_i)(y)$ for every $y \in Q$. Thus we get that

$$f(x_n)f(x_{n-1})\dots f(x_1)(\bot) \leq f(x_i)^n(\bot),$$

thus every member in the set $\{f(x_n)f(x_{n-1})\dots f(x_1)(\bot) : x_1,\dots,x_n \in X\}$ has an upper bound in its subset $\{f(x)^n(\bot) : x \in X\}$, hence we get

$$\bigvee_{x_1,\dots,x_n\in X} f(x_n)f(x_{n-1})\dots f(x_1)(\bot) = \bigvee_{x\in X} f(x)^n(\bot).$$

Thus,

$$\bigvee_{n<\omega}\bigvee_{x_1,\dots,x_n\in X} f(x_n)f(x_{n-1})\dots f(x_1)(\bot) = \bigvee_{n<\omega}\bigvee_{x\in X} f(x)^n(\bot) = \bigvee_{x\in X}\bigvee_{n<\omega} f(x)^n(\bot),$$

and as the left-hand side is $\mathsf{lfp}(f)(x^*)$, the right-hand side is $\bigvee_{x\in X}\mathsf{lfp}(f)(x)$ due to the Tarski Fixed Point Theorem, the statement is proved.

Thus we can finally define the semantics of the recursion operator as

$$[[\Gamma \ \vdash \ Y_\sigma(M) \ : \ \sigma]](d_1,\dots,d_n) = \mathsf{lfp}\Big([[\Gamma \vdash M : \sigma \to \sigma]]\Big)(d_1,\dots,d_n).$$

Applying the induction hypothesis we get that $[[\Gamma \vdash M : \sigma \to \sigma]]$ is a $\Big[[[\Gamma]] \to \big[[[\sigma]] \to [[\sigma]]\big]\Big]$ function, thus its lfp is a $\Big[[[\Gamma]] \to [[\sigma]]\Big]$ function, as intended.

Equivalence of the two semantics

We have defined two semantics for terms: the operational semantics (defined over pure terms) and the denotational semantics (defined for terms).

One part of the two semantics being equivalent is the following:

If $\Gamma \vdash M : \sigma$ is a term and $M \triangleright N$, then $[[\Gamma \vdash M : \sigma]] = [[\Gamma \vdash N : \sigma]]$.

This of course has the corollary that if $\Gamma \vdash M : \sigma$ and $M \triangleright^* V$, in particular if $M \Downarrow V$, then $[[\Gamma \vdash M : \sigma]] = [[\Gamma \vdash V : \sigma]]$ as well.

But this is not enough for an exact coincidence: for example, the term $M = (Y(\lambda f.\lambda x.f(x)))(0)$ enters an infinite loop according to the operational semantics as:

$$\begin{aligned} & (Y(\lambda f.\lambda x.f(x)))(0) \\ \triangleright\ & ((\lambda f.\lambda x.f(x))(Y(\lambda f.\lambda x.f(x))))(0) \\ \triangleright\ & (\lambda x.(Y(\lambda f.\lambda x.f(x)))(x))(0) \\ \triangleright\ & (Y(\lambda f.\lambda x.f(x)))(0) \end{aligned}$$

and the computation loops over these three terms forever. Thus, if the two semantics "coincide" in a sense, then this term should have the value $\bot$ (corresponding to the infinite loop). This property can be captured by the following statement:

If $\vdash M : \mathsf{nat}$ is a term, then for any $n \in \mathbb{N}$, $M \Downarrow n$ if and only if $[[\vdash M : \mathsf{nat}]]$ is the constant n function.

Also, M does not have a value if and only if $[[\vdash M : \mathsf{nat}]]$ is the constant $\bot$ function.

In this section we show that both properties indeed hold.

The first statement is technically easy to check but of course it has many cases. First we show that term substitution factors through denotational semantics, which will be used in the λ-abstraction case.

Proposition

If $\Gamma \vdash M : \sigma$ for $\Gamma = x_1 : \sigma_1, \dots, x_n : \sigma_n$ and for each $1 \leq i \leq n$, $\Delta \vdash N_i : \sigma_i$, then

$$[[\Gamma \vdash M : \sigma]] \circ \Big\langle [[\Delta \vdash N_1 : \sigma_1]], \dots, [[\Delta \vdash N_n : \sigma_n]] \Big\rangle = [[\Delta \vdash M[\vec{x}/\vec{N}] : \sigma]].$$

Proof

We show the claim by induction on the structure of M.

1. If $M = 0$ (and thus $\sigma = \mathsf{nat}$), then $M[\vec{x}/\vec{N}] = 0$ as well and both sides evaluate to the constant 0 function.
2. If $M = x_i$ (and thus $\sigma = \sigma_i$), then the left-hand side is $[[\Delta \vdash N_i : \sigma_i]]$ (as $[[\Gamma \vdash x_i : \sigma_i]]$ is the projection to the ith coordinate), and since $x_i[\vec{x}/\vec{N}] = N_i$, the right-hand side is $[[\Delta \vdash N_i : \sigma_i]]$ as well.
3. If $M = \mathrm{op}(M_1, \dots, M_k)$ for $\Gamma \vdash M_j : \tau_j$ such that

$$[[\Gamma \vdash \mathrm{op}(M_1, \dots, M_k) : \sigma]] = [[\mathrm{op}]] \circ \langle [[\Gamma \vdash M_1 : \tau_1]], \dots, [[\Gamma \vdash M_k : \tau_k]] \rangle$$

for some function $[[\mathrm{op}]]$ and $M[\vec{x}/\vec{N}] = \mathrm{op}(M_1[\vec{x}/\vec{N}], \dots, M_k[\vec{x}/\vec{N}])$, then

$$\begin{aligned}
& [[\Gamma \vdash M : \sigma]] \circ \Big\langle [[\Delta \vdash N_1 : \sigma_1]], \dots, [[\Delta \vdash N_n : \sigma_n]] \Big\rangle \\
= \; & [[\mathrm{op}]] \circ \Big\langle [[\Gamma \vdash M_1 : \tau_1]], \dots, [[\Gamma \vdash M_k : \tau_k]] \Big\rangle \circ \Big\langle [[\Delta \vdash N_1 : \sigma_1]], \dots, [[\Delta \vdash N_n : \sigma_n]] \Big\rangle \\
\overset{*}{=} \; & [[\mathrm{op}]] \circ \Big\langle [[\Delta \vdash M_1[\vec{x}/\vec{N}] : \tau_1]], \dots, [[\Delta \vdash M_k[\vec{x}/\vec{N}] : \tau_k]] \Big\rangle \\
= \; & [[\Delta \vdash \mathrm{op}(M_1[\vec{x}/\vec{N}], \dots, M_k[\vec{x}/\vec{N}]) : \sigma]] \\
= \; & [[\Delta \vdash M[\vec{x}/\vec{N}] : \sigma]]
\end{aligned}$$

where in the starred step we applied the induction hypothesis for the terms M_i. This handles the case of succ, pred, ifzero, $M(N)$ and $Y_\sigma(M)$, with $[[\mathrm{op}]]$ being $[[\mathsf{succ}]]$, $[[\mathsf{pred}]]$, $[[\mathsf{ifzero}]]$, eval and lfp, respectively.

4. Finally, if $M = \lambda x : \tau_1.M_1$ and thus $\sigma = \tau_1 \to \tau_2$, with x not appearing in Γ, then

$$\begin{aligned}
& [[\Gamma \vdash \lambda x : \tau_1.M_1 \;:\; \tau_1 \to \tau_2]] \circ \Big\langle [[\Delta \vdash N_1 : \sigma_1]], \dots, [[\Delta \vdash N_n : \sigma_n]] \Big\rangle \\
= \; & \mathsf{curry}[[\Gamma, x : \tau_1 \vdash M_1 : \tau_2]] \circ \Big\langle [[\Delta \vdash N_1 : \sigma_1]], \dots, [[\Delta \vdash N_n : \sigma_n]] \Big\rangle \\
= \; & \mathsf{curry}[[\Gamma, z : \tau_1 \vdash M_1[x/z] : \tau_2]] \circ \Big\langle [[\Delta \vdash N_1 : \sigma_1]], \dots, [[\Delta \vdash N_n : \sigma_n]] \Big\rangle
\end{aligned}$$

for any variable z. For the right side,

$$\begin{aligned}
& [[\Delta \vdash (\lambda x : \tau_1.M_1)[\vec{x}/\vec{N}] : \tau_1 \to \tau_2]] \\
= \; & [[\Delta \vdash \lambda z : \tau_1.M_1[x/z][\vec{x}/\vec{N}] \;:\; \tau_1 \to \tau_2]] \\
= \; & \mathsf{curry}[[\Delta, z : \tau_1 \vdash M_1[x/z][\vec{x}/\vec{N}] : \tau_2]]
\end{aligned}$$

for the fresh variable z not occurring in any of the N_i. Both results are functions of type $[[\Delta]] \to \Big([[\tau_1]] \to [[\tau_2]]\Big)$: such functions f and g coindide if and only if $f(\vec{d})(d) = g(\vec{d})(d)$ for each $\vec{d} \in [[\Delta]]$ and $d \in [[\tau_1]]$. Thus, as

$$\begin{aligned}
& \Big(\mathsf{curry}[[\Gamma, z : \tau_1 \vdash M_1[x/z] : \tau_2]] \circ \Big\langle [[\Delta \vdash N_1 : \sigma_1]], \dots, [[\Delta \vdash N_n : \sigma_n]] \Big\rangle\Big)(\vec{d})(d) \\
= \; & \mathsf{curry}[[\Gamma, z : \tau_1 \vdash M_1[x/z] : \tau_2]]\Big([[\Delta \vdash N_1 : \sigma_1]](\vec{d}), \dots, [[\Delta \vdash N_n : \sigma_n]](\vec{d})\Big)(d) \\
= \; & [[\Gamma, z : \tau_1 \vdash M_1[x/z] : \tau_2]]\Big([[\Delta \vdash N_1 : \sigma_1]](\vec{d}), \dots, [[\Delta \vdash N_n : \sigma_n]](\vec{d}), d\Big) \\
= \; & [[\Gamma, z : \tau_1 \vdash M_1[x/z] : \tau_2]]\Big(\\
& [[\Delta, z : \tau_1 \vdash N_1 : \sigma_1]](\vec{d}, d), \dots, [[\Delta, z : \tau_1 \vdash N_n : \sigma_n]](\vec{d}, d), [[\Delta, z : \tau_1 \vdash z : \tau_1]](\vec{d}, d)\Big) \\
\overset{*}{=} \; & [[\Delta, z : \tau_1 \vdash M_1[x/z][\vec{x}/\vec{N}, z/z] : \tau_2]](\vec{d}, d) \\
= \; & \mathsf{curry}[[\Delta, z : \tau_1 \vdash M_1[x/z][\vec{x}/\vec{N}] : \tau_2]](\vec{d})(d)
\end{aligned}$$

and the claim is proved.

We are ready to show one direction of the semantics equivalence:

Proposition

If $\Gamma \vdash M : \sigma$ is a term and $M \triangleright N$, then $[[\Gamma \vdash M : \sigma]] = [[\Gamma \vdash N : \sigma]]$.

Proof

We apply induction on the structure of M, having many subcases.

1. If $M = x$ or $M = 0$ or $M = \lambda x : \tau.M'$, then there is no N with $M \triangleright N$, so the claim holds.

2. If $M = (M_1)(M_2)$ and $M_1 = \lambda x : \tau.M_1'$, then $\Gamma \vdash M_1 : \tau \to \sigma$ and $\Gamma \vdash M_2 : \tau$ are terms. Now if $M \triangleright N$, then $N = M_1'[x/M_2]$. We have to show that $[[\Gamma \vdash (\lambda x : \tau.M_1')(M_2) : \sigma]] = [[\Gamma \vdash M_1'[x/M_2] : \sigma]]$. Spelling out the definitions,

$$\begin{aligned} & [[\Gamma \vdash (\lambda x : \tau.M_1')(M_2)]] \\ = \ & \mathsf{eval} \circ \left\langle [[\Gamma \vdash \lambda x : \tau.M_1' : \tau \to \sigma]], [[\Gamma \vdash M_2 : \tau]] \right\rangle \\ = \ & \mathsf{eval} \circ \left\langle \mathsf{curry}[[\Gamma, x : \tau \vdash M_1' : \sigma]], [[\Gamma \vdash M_2 : \tau]] \right\rangle. \end{aligned}$$

   For an input $\vec{d} \in [[\Gamma]]$ this is evaluated as

$$\begin{aligned} & \mathsf{eval} \circ \left\langle \mathsf{curry}[[\Gamma, x : \tau \vdash M_1' : \sigma]], [[\Gamma \vdash M_2 : \tau]] \right\rangle (\vec{d}) \\ = \ & \mathsf{curry}[[\Gamma, x : \tau \vdash M_1' : \sigma]](\vec{d})([[\Gamma \vdash M_2 : \tau]](\vec{d})) \\ = \ & [[\Gamma, x : \tau_1 \vdash M_1' : \sigma]](\vec{d}, [[\Gamma \vdash M_2 : \tau]](\vec{d})) \\ = \ & [[\Gamma \vdash M_1'[x/M_2] : \sigma]](\vec{d}). \end{aligned}$$

3. If $M = (M_1)(M_2)$ and $M_1 \triangleright M_1'$, then by $M \triangleright N$ it has to be the case $N = (M_1')(M_2)$. By the induction hypothesis, $[[\Gamma \vdash M_1 : \tau \to \sigma]] = [[\Gamma \vdash M_1' : \tau \to \sigma]]$ and thus

$$\begin{aligned} [[\Gamma \vdash M : \sigma]] &= \mathsf{eval}([[\Gamma \vdash M_1 : \tau \to \sigma]], [[\Gamma \vdash M_2 : \tau]]) \\ &= \mathsf{eval}([[\Gamma \vdash M_1' : \tau \to \sigma]], [[\Gamma \vdash M_2 : \tau]]) \\ &= [[\Gamma \vdash N : \sigma]]. \end{aligned}$$

4. If $M = \mathsf{succ}(M_1)$, then $N = \mathsf{succ}(N_1)$ for some $M_1 \triangleright N_1$. By the induction hypothesis, $[[\Gamma \vdash M_1 : \mathsf{nat}]] = [[\Gamma \vdash N_1 : \mathsf{nat}]]$, thus

$$[[\Gamma \vdash M : \mathsf{nat}]] = [[\mathsf{succ}]] \circ [[\Gamma \vdash M_1 : \mathsf{nat}]] = [[\mathsf{succ}]] \circ [[\Gamma \vdash N_1 : \mathsf{nat}]] = [[\Gamma \vdash N : \mathsf{nat}]].$$

5. If $M = \mathsf{pred}(0)$, then $N = 0$ and we have

$$[[\Gamma \vdash \mathsf{pred}(0) : \mathsf{nat}]] = [[\mathsf{pred}]] \circ [[\Gamma \vdash 0 : \mathsf{nat}]]$$

   which is $[[\mathsf{pred}]]$ applied on the constant 0, that is, indeed the constant 0 function.

6. If $M = \mathsf{pred}(\mathsf{succ}^{n+1}(0))$, then $N = \mathsf{succ}^n(0)$. As we already argued, $[[\Gamma \vdash \mathsf{succ}^n(0) : \mathsf{nat}]]$ is the constant n function for each n. As $[[pred]]$ decreases the positive values by one, we get $[[\Gamma \vdash M : \mathsf{nat}]]$ is also the constant n function.

7. If $M = \mathsf{pred}(M_1)$, with $M_1 \rhd N_1$, then $N = \mathsf{succ}(N_1)$. By the induction hypothesis, $[[\Gamma \vdash M_1 : \mathsf{nat}]] = [[\Gamma \vdash N_1 : \mathsf{nat}]]$, thus

$$[[\Gamma \vdash M : \mathsf{nat}]] = [[\mathsf{pred}]] \circ [[\Gamma \vdash M_1 : \mathsf{nat}]] = [[\mathsf{pred}]] \circ [[\Gamma \vdash N_1 : \mathsf{nat}]] = [[\Gamma \vdash N : \mathsf{nat}]].$$

8. If $M = \mathsf{ifzero}(0, M_2, M_3)$, then $N = M_2$. Then for any $d_1, \ldots, d_n$ we have

$$[[\Gamma \vdash M : \mathsf{nat}]](d_1, \ldots, d_n) = [[\mathsf{ifzero}]](v_1, v_2, v_3)$$

with $v_i = [[\Gamma \vdash M_i : \mathsf{nat}]](d_1, \ldots, d_n)$ where $M_1 = 0$. As $[[\Gamma \vdash 0 : \mathsf{nat}]]$ is the constant zero function, we have that $v_1 = 0$, thus by the definition of $[[\mathsf{ifzero}]]$, $[[\mathsf{ifzero}]](v_1, v_2, v_3) = v_2$ which is exactly $[[\Gamma \vdash M_2 : \mathsf{nat}]]$, as intended.

9. If $M = \mathsf{ifzero}(n+1, M_2, M_3)$, then $N = M_3$. Then for any $d_1, \ldots, d_n$ we have

$$[[\Gamma \vdash M : \mathsf{nat}]](d_1, \ldots, d_n) = [[\mathsf{ifzero}]](v_1, v_2, v_3)$$

with $v_i = [[\Gamma \vdash M_i : \mathsf{nat}]](d_1, \ldots, d_n)$ where $M_1 = n+1$. As $[[\Gamma \vdash n+1 : \mathsf{nat}]]$ is the constant $n+1$ function, we have that $v_1 = n+1$, thus by the definition of $[[\mathsf{ifzero}]]$, $[[\mathsf{ifzero}]](v_1, v_2, v_3) = v_3$ which is exactly $[[\Gamma \vdash M_3 : \mathsf{nat}]]$, as intended.

10. If $M = \mathsf{ifzero}(M_1, M_2, M_3)$ with $M_1 \rhd M_1'$, then $N = \mathsf{ifzero}(M_1', M_2, M_3)$. By the induction hypothesis, $[[\Gamma \vdash M_1 : \mathsf{nat}]] = [[\Gamma \vdash M_1' : \mathsf{nat}]]$, thus

$$\begin{aligned} & [[\Gamma \vdash \mathsf{ifzero}(M_1, M_2, M_3) : \mathsf{nat}]] \\ = \; & [[\mathsf{ifzero}]] \circ \langle [[\Gamma \vdash M_1 : \mathsf{nat}]], [[\Gamma \vdash M_2 : \mathsf{nat}]], [[\Gamma \vdash M_3 : \mathsf{nat}]] \rangle \\ = \; & [[\mathsf{ifzero}]] \circ \langle [[\Gamma \vdash M_1' : \mathsf{nat}]], [[\Gamma \vdash M_2 : \mathsf{nat}]], [[\Gamma \vdash M_3 : \mathsf{nat}]] \rangle \\ = \; & [[\Gamma \vdash \mathsf{ifzero}(M_1', M_2, M_3) : \mathsf{nat}]] \\ = \; & [[\Gamma \vdash N : \mathsf{nat}]]. \end{aligned}$$

11. Finally, if $M = Y_\sigma(M_1)$, then $N = M_1(Y_\sigma(M_1))$. Then, $\Gamma \vdash M_1 : \sigma \to \sigma$ is a term. Let f stand for the function $[[\Gamma \vdash M_1 : \sigma \to \sigma]]$. Then $[[\Gamma \vdash M : \sigma]](d_1, \ldots, d_n)$ is $\mathsf{lfp}(f)(d_1, \ldots, d_n)$ and $[[\Gamma \vdash N : \sigma]](d_1, \ldots, d_n)$ is $\mathsf{eval}\Big(f(d_1, \ldots, d_n), \mathsf{lfp}(f)(d_1, \ldots, d_n)\Big)$. By definition of lfp, $\mathsf{lfp}(f)(d_1, \ldots, d_n)$ is a (actually, the least) fixed point of the function $f(d_1, \ldots, d_n)$ (which is a $[[\sigma]] \to [[\sigma]]$ continuous function). Hence, $\mathsf{eval}\Big(f(d_1, \ldots, d_n), \mathsf{lfp}(f)(d_1, \ldots, d_n)\Big)$ is $f(d_1, \ldots, d_n)\Big(\mathsf{lfp}(f)(d_1, \ldots, d_n)\Big) = \mathsf{lfp}(f)(d_1, \ldots, d_n)$, i.e., $[[\Gamma \vdash M : \sigma]](d_1, \ldots, d_n)$, thus the two functions coincide here as well.

Now we turn to the our direction. We want to prove that whenever

$$[[\vdash M : \mathsf{nat}]] = n$$

for some $n \in \mathbb{N}$, then

$$M \Downarrow \underline{n}.$$

In order to show this, we have to prove some more correlations between the two semantics. We define for each Γ and σ a relation denoted $\leq$ between members of $[[\Gamma]] \to [[\sigma]]$ and terms $\Gamma \vdash M : \sigma$ inductively as follows.

Definition

Let $\Gamma \vdash M : \sigma$ be a term and $f : [[\Gamma]] \to [[\sigma]]$ be a function.

We define $f \leq (\Gamma \vdash M : \sigma)$ for the context $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ inductively on the structure of σ and the cardinality of Γ as follows:

- If $\sigma = \mathsf{nat}$, then $f \leq \Gamma \vdash M : \mathsf{nat}$ if for each $d_i \in [[\sigma_i]]$, $\vdash N_i : \sigma_i$ for $i = 1, \ldots, n$ we either have $f(\vec{d}) = \bot$, or $f(\vec{d}) = n \in \mathbb{N}$ and $M[\vec{x}/\vec{N}] \Downarrow \underline{n}$.
- If $\sigma = \tau_1 \to \tau_2$, then $f \leq \Gamma \vdash M : \tau_1 \to \tau_2$ if for each $d_i \in [[\sigma_i]]$, $d \in [[\tau_1]]$, $\vdash N_i : \sigma_i$ with $d_i \leq N_i$ and $\vdash N : \tau_1$ with $d \leq N$ we have $f(\vec{d})(d) \leq (M[\vec{x}/\vec{N}])(N)$.

We will prove that for any term,

$$[[\Gamma \vdash M : \sigma]] \;\leq\; \Gamma \vdash M : \sigma.$$

This in particular implies that $[[\vdash M : \mathsf{nat}]] \;\leq \vdash M : \mathsf{nat}$, which by the definition of the base case implies that if $[[\vdash M : \mathsf{nat}]] = n \in \mathbb{N}$, then $M \Downarrow \underline{n}$, and that's what we are seeking for.

Before the proof itself, we show several handy statements, needed mainly for the case of the recursion operator.

Proposition

Assume $f \leq f' \leq \Gamma \vdash M : \sigma$. Then $f \leq \Gamma \vdash M : \sigma$.

Thus in particular, $\bot \leq \Gamma \vdash M : \sigma$ for the least element $\bot$ of the corresponding CPO.

Proof

Let $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ and $d_i \leq N_i$ for $d_i \in [[\sigma_i]]$ and $\vdash N_i : \sigma_i$. We use induction on the structure of σ.

If $\sigma = \mathsf{nat}$, then $f(\vec{d})$ is either $\bot$, or some $n \in \mathbb{N}$. If it is $\bot$, then it's fine. Otherwise, $f(\vec{d}) = n$. By $f \leq f'$ it has to be the case $f'(\vec{d}) = n$ as well. We get $M[\vec{x}/\vec{N}] \Downarrow n$ by $f' \leq \Gamma \vdash M : \sigma$.

If $\sigma = \sigma_1 \to \sigma_2$, then we have to show that whenever $d \leq N$ for $d \in [[\sigma_1]]$ and $\vdash N : \sigma_1$, it also holds that

$$f(\vec{d})(d) \;\leq\; \vdash (M[\vec{x}/\vec{N}])(N) : \sigma_2.$$

Since $f \leq f'$ we have $f(\vec{d}) \leq f'(\vec{d})$, thus $f(\vec{d})(d) \leq f'(\vec{d})(d)$ and by $f' \leq \Gamma \vdash M : \sigma$ we have

$$f'(\vec{d})(d) \;\leq\; \vdash (M[\vec{x}/\vec{N}])(N) : \sigma_2.$$

Applying the induction hypothesis we get that $f(\vec{d})(d) \;\leq\; \vdash (M[\vec{x}/\vec{N}])(N) : \sigma_2$, exactly what we needed.

Proposition

Assume $f \leq \vdash M : \sigma$ and $N \triangleright M$. Then $f \leq \vdash N : \sigma$.

Proof

We use induction on the structure of σ.

If $\sigma = \mathsf{nat}$, then by $f \trianglelefteq\vdash M : \mathsf{nat}$, we either have $f = \bot$, in which case $f \trianglelefteq\vdash N : \mathsf{nat}$ as well, or $f = n \in \mathbb{N}$, in which case $M \Downarrow \underline{n}$. Thus, by $N \triangleright M$ we also have $N \Downarrow \underline{n}$, hence $f \trianglelefteq\vdash N : \mathsf{nat}$.

If $\sigma = \sigma_1 \to \sigma_2$, then let $d \in [[\sigma_1]]$ and $\vdash K : \sigma_1$ with $d \trianglelefteq K$. By $f \trianglelefteq\vdash M : \sigma_1 \to \sigma_2$ we get that $f(d) \trianglelefteq\vdash M(K) : \sigma_2$. From $N \triangleright M$ we also get $N(K) \triangleright M(K)$. Thus applying the induction hypothesis we get $f(d) \trianglelefteq N(K)$, which proves $f \trianglelefteq\vdash N : \sigma$.

Proposition

Assume $\Gamma \vdash M : \sigma$ is a term and $F = \{f_i : i \in I\}$ is a linearly ordered set of functions $[[\Gamma]] \to [[\sigma]]$ with $f_i \trianglelefteq \Gamma \vdash M : \sigma$ for each $i \in I$. Then $\bigvee F \trianglelefteq \Gamma \vdash M : \sigma$ as well.

Proof

We use induction on the structure of σ. Let $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$, and for each $1 \leq i \leq n$, let $d_i \in [[\sigma_i]]$, $\vdash N_i : \sigma_i$ with $d_i \trianglelefteq N_i$.

Clearly, $(\bigvee F)(\vec{d}) = \bigvee_{f \in F} f(\vec{d})$ and the set $F' = \{f(\vec{d}) : f \in F\}$ is a linearly ordered subset of $[[\sigma]]$.

If $\sigma = \mathsf{nat}$, then either $\bigvee F' = \bot$ which is fine, or $\bigvee F' = n$ for some $n \in \mathbb{N}$. In the latter case there has to be a function $f \in F$ with $f(\vec{d}) = n$. By $f \trianglelefteq \Gamma \vdash M : \mathsf{nat}$ we get that $M[\vec{x}/\vec{N}] \Downarrow \underline{n}$. Thus $\bigvee F \trianglelefteq \Gamma \vdash M : \mathsf{nat}$.

If $\sigma = \sigma_1 \to \sigma_2$, then we have to show that whenever $d \trianglelefteq N$ for $d \in [[\sigma_1]]$ and $\vdash N : \sigma_1$, then $\left(\bigvee_{f \in F} f(\vec{d})\right)(d) \trianglelefteq \Gamma \vdash M[\vec{x}/\vec{N}](N)$.

But as $\left(\bigvee_{f \in F} f(\vec{d})\right)(d) = \bigvee_{f \in F}(f(\vec{d})(d))$ and since F is linearly ordered, so is $\{f(\vec{d})(d) : f \in F\}$. Also, since by assumption $f \trianglelefteq \Gamma \vdash M : \sigma_1 \to \sigma_2$ for each $f \in F$, we have that $f(\vec{d})(d) \trianglelefteq\vdash M[\vec{x}/\vec{N}](N)$. Since these objects form a linearly ordered subset of $[[\sigma_2]]$, we can apply induction and get that $\bigvee_{f \in F}(f(\vec{d})(d)) \trianglelefteq \ \vdash M[\vec{x}/\vec{N}](N)$, proving the claim.

Proposition

For any term $\Gamma \vdash M : \sigma$ it holds that

$$[[\Gamma \vdash M : \sigma]] \ \trianglelefteq \ \Gamma \vdash M : \sigma.$$

Proof

We apply induction on the structure of M, as usual. Let Γ be the context $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ and for each $i = 1, \ldots, n$ let $d_i \in [[\sigma_i]]$ and $\vdash N_i : \sigma_i$ with $d_i \trianglelefteq N_i$.

1. If $M = 0$, then $\sigma = \mathsf{nat}$ and we have $[[\Gamma \vdash 0 : \mathsf{nat}]](\vec{d_i}) = 0$, also $0[\vec{x}/\vec{N_i}] = 0$, and $0 \Downarrow \underline{0}$ indeed holds.
2. If $M = x_i$, then $\sigma = \sigma_i$ and $[[\Gamma \vdash x_i : \sigma_i]](\vec{d}) = d_i$. Also, $x_i[\vec{x}/\vec{N}] = N_i$, hence we have to show $d_i \trianglelefteq \Gamma \vdash N_i : \sigma_i$ which is granted by our assumption on the d_is and

N_is.

3. If $M = \mathsf{succ}(K)$, then $\sigma = \mathsf{nat}$. Then $[[\Gamma \vdash M : \mathsf{nat}]](\vec{d})$ is $[[\mathsf{succ}]]\Big([[\Gamma \vdash K : \mathsf{nat}\Big]](\vec{d})\big)$. Now if $[[\Gamma \vdash K : \mathsf{nat}\Big]](\vec{d}) = \bot$, then this result is also $\bot$; if it's some $n \in \mathbb{N}$, then the result is $n+1$. Applying the induction hypothesis on $([[\Gamma \vdash K : \mathsf{nat}\Big]](\vec{d}) = n$ we get that $K[\vec{x}/\vec{N}] \Downarrow \underline{n}$. Thus $M[\vec{x}/\vec{N}] = \mathsf{succ}(K[\vec{x}/\vec{N}]) \Downarrow \underline{n+1}$.

4. If $M = \mathsf{pred}(K)$, then $\sigma = \mathsf{nat}$ and $[[\Gamma \vdash M : \mathsf{nat}]](\vec{d})$ is $[[\mathsf{pred}]]\Big([[\Gamma \vdash K : \mathsf{nat}\Big]](\vec{d})\big)$. Let x denote $[[\Gamma \vdash K : \mathsf{nat}\Big]](\vec{d}) \in \mathbb{N}_\bot$. If $x = \bot$, then $[[\Gamma \vdash M : \mathsf{nat}]](\vec{d}) = [[\mathsf{pred}]](\bot) = \bot$ as well. If $x = 0$, then $[[\Gamma \vdash M : \mathsf{nat}]](\vec{d}) = [[\mathsf{pred}]](0) = 0$. Applying the induction hypothesis we get that $K[\vec{x}/\vec{N}] \Downarrow 0$. Thus, $M[\vec{x}/\vec{N}] = \mathsf{pred}(K[\vec{x}/\vec{N}]) \rhd^* \mathsf{pred}(0) \rhd 0$, so $M \Downarrow 0$.

   Finally, if $x = n+1$ for some $n \in \mathbb{N}$, then $[[\Gamma \vdash M : \mathsf{nat}]](\vec{d}) = [[\mathsf{pred}]](n+1) = n$. Applying the induction hypothesis on K we get that $K[\vec{x}/\vec{N}] \Downarrow \underline{n}$. Hence $M[\vec{x}/\vec{N}] = \mathsf{pred}(K[\vec{x}/\vec{N}]) \rhd^* \mathsf{pred}(\underline{n+1}) \rhd \underline{n}$, thus $M[\vec{x}/\vec{N}] \Downarrow \underline{n}$.

5. Assume $M = \mathsf{ifzero}(M_1, M_2, M_3)$. Then $\sigma = \mathsf{nat}$ again. Now

   $$[[\Gamma \vdash M : \mathsf{nat}]](\vec{d}) = [[\mathsf{ifzero}]]([[\Gamma \vdash M_1 : \mathsf{nat}]](\vec{d}), [[\Gamma \vdash M_2 : \mathsf{nat}]](\vec{d}), [[\Gamma \vdash M_3 : \mathsf{nat}]](\vec{d})).$$

   Let v_i stand for $[[\Gamma \vdash M_i : \mathsf{nat}]](\vec{d})$.

   If $v_1 = \bot$, then the result is $\bot$ which is fine.

   If $v_1 = 0$, then the result is v_2. If $v_2 = \bot$, it's fine. Otherwise, let $v_2 = n \in \mathbb{N}$. Applying the induction hypothesis we get from $[[\Gamma \vdash M_1 : \mathsf{nat}]] \le \Gamma \vdash M_1 : \mathsf{nat}$ that $M_1[\vec{x}/\vec{N}] \Downarrow 0$. Thus, $M[\vec{x}/\vec{N}] = \mathsf{ifzero}(M_1[\vec{x}/\vec{N}], M_2[\vec{x}/\vec{N}], M_3[\vec{x}/\vec{N}]) \rhd^* \mathsf{ifzero}(0, M_2[\vec{x}/\vec{N}], M_3[\vec{x}/\vec{N}]) \rhd M_2[\vec{x}/\vec{N}]$. Now applying the induction hypothesis we get from $[[\Gamma \vdash M_2 : \mathsf{nat}]] \le \Gamma \vdash M_2 : \mathsf{nat}$ that $M_2[\vec{x}/\vec{N}] \Downarrow n$. Thus, $M[\vec{x}/\vec{N}] \Downarrow n$.

   Finally if v_1 is a positive integer $m+1$, then the result is v_3. If $v_3 = \bot$, it's fine. Otherwise, let $v_3 = n \in \mathbb{N}$. Applying the induction hypothesis we get from $[[\Gamma \vdash M_1 : \mathsf{nat}]] \le \Gamma \vdash M_1 : \mathsf{nat}$ that $M_1[\vec{x}/\vec{N}] \Downarrow \underline{m+1}$. Thus, $M[\vec{x}/\vec{N}] = \mathsf{ifzero}(M_1[\vec{x}/\vec{N}], M_2[\vec{x}/\vec{N}], M_3[\vec{x}/\vec{N}]) \rhd^* \mathsf{ifzero}(\underline{m+1}, M_2[\vec{x}/\vec{N}], M_3[\vec{x}/\vec{N}]) \rhd M_3[\vec{x}/\vec{N}]$. Now applying the induction hypothesis we get from $[[\Gamma \vdash M_3 : \mathsf{nat}]] \le \Gamma \vdash M_3 : \mathsf{nat}$ that $M_3[\vec{x}/\vec{N}] \Downarrow n$. Thus, $M[\vec{x}/\vec{N}] \Downarrow n$.

6. Assume $M = N(K)$. Then $\Gamma \vdash N : \tau \to \sigma$ and $\Gamma \vdash K : \tau$. Then,

   $$\begin{aligned} & [[\Gamma \vdash N(K) : \sigma]](\vec{d}) \\ = \;& [[\Gamma \vdash N : \tau \to \sigma]](\vec{d})\Big([[\Gamma \vdash K : \tau]](\vec{d})\Big). \end{aligned}$$

   Applying the induction hypothesis for N, we get that whenever $d \le K'$ for some $d \in [[\tau]]$ and $\vdash K' : \tau$, then $[[\Gamma \vdash N : \tau \to \sigma]](\vec{d})(d) \le N[\vec{x}/\vec{N}](K')$. Applying the induction hypothesis for K, we get that $d = [[\Gamma \vdash K : \tau]](\vec{d}) \le K[\vec{x}/\vec{N}] = K'$, thus we have that $[[\Gamma \vdash N(K) : \sigma]](\vec{d}) \le N[\vec{x}/\vec{N}](K[\vec{x}/\vec{N}]) = M[\vec{x}/\vec{N}]$.

7. Assume $M = \lambda x : \sigma_1.N : \sigma_1 \to \sigma_2$. We have to show that whenever $d \leq K$ for $d \in [[\sigma_1]]$ and $\vdash K : \sigma_1$, then

$$[[\Gamma \vdash \lambda x : \sigma_1.N \;:\; \sigma_1 \to \sigma_2]](\vec{d})(d) \leq (\lambda x : \sigma_1.N)[\vec{x}/\vec{N}](K).$$

The left-hand side equals to $[[\Gamma, x : \sigma_1 \vdash N : \sigma_2]](\vec{d}, d)$, and the right-hand side is $(\lambda z : \sigma_1.N[x/z][\vec{x}/\vec{N}])(K) = N[\vec{x}/\vec{N}, x/K]$ for some fresh variable z, for which we can apply the induction hypothesis.

8. Finally, assume $M = Y_\sigma(N)$. Then $\Gamma \vdash N : \sigma \to \sigma$. Let f stand for the function $[[\Gamma \vdash N : \sigma \to \sigma]]$. Applying the induction hypothesis, $f \leq \Gamma \vdash N : \sigma \to \sigma$, which is function type, so this means that $f(\vec{d})(d) \leq N[\vec{x}/\vec{N}](K)$ for any $d \leq K$, $d \in [[\sigma]]$, $\vdash K : \sigma$. We have to show that $[[\Gamma \vdash Y_\sigma(N) : \sigma]](\vec{d}) \leq \Gamma \vdash Y_\sigma(N[\vec{x}/\vec{N}]) : \sigma$.

   From the Tarski Fixed Point Theorem we know that

$$[[\Gamma \vdash Y_\sigma(N) : \sigma]](\vec{d}) = \bigvee_{n \geq 0} \left(f(\vec{d})\right)^n(\bot).$$

   We prove that $\left(f(\vec{d})\right)^n(\bot) \leq Y_\sigma(N[\vec{x}/\vec{N}]) : \sigma$ which shows our claim by the previous proposition.

   We also know that $\bot \leq \Gamma \vdash Y_\sigma(N[\vec{x}/\vec{N}]) : \sigma$, handling the case $n = 0$. We now apply induction: if $\left(f(\vec{d})\right)^n(\bot) \leq Y_\sigma(N[\vec{x}/\vec{N}])$, then by $f \leq \Gamma \vdash N : \sigma \to \sigma$ we get

$$\left(f(\vec{d})\right)^{n+1}(\bot) = f(\vec{d})(\left(f(\vec{d})\right)^n(\bot)) \leq N[\vec{x}/\vec{N}](Y_\sigma(N[\vec{x}/\vec{N}])).$$

   Since $Y_\sigma(N[\vec{x}/\vec{N}]) \triangleright N[\vec{x}/\vec{N}](Y_\sigma(N[\vec{x}/\vec{N}]))$, the claim is proved.

Summing up, in the second part we

- defined the syntax of λ-calculus, which is the mathematical model of pure functional programs;
- defined the operational semantics of λ-terms, which is the „how” part of the computation, a stepwise rewriting rule set;
- defined the denotational semantics of λ-terms, which is the „what” view of the computation, assigning continuous functions to the terms;
- and proved that the two semantics exactly correspond to each other.

the end.